

DRAFT

APPLICATION SECURITY

DEVELOPER'S GUIDE



Version 1.0

October 4, 2002

Applications and Computing Security Division
Center for Information Assurance Applications
5275 Leesburg Pike
Falls Church, VA 22041

(This document is for review. Comments, if any, can be sent to

JainD@ncr.disa.mil or KoehlerS@ncr.disa.mil)

FOR INFORMATIONAL PURPOSES

TABLE OF CONTENTS

	Page Number
1.0 INTRODUCTION	1
1.1 PURPOSE.....	1
1.2 SCOPE.....	2
1.2.1 Subjects Not Addressed in This Document.....	3
1.3 INTENDED AUDIENCE.....	3
1.4 NOTE ON STYLE	4
2.0 BACKGROUND.....	5
2.1 ORGANIZATION AND CONTENT OF THIS DOCUMENT	5
2.2 HOW TO USE THIS DOCUMENT.....	6
3.0 WHAT IS WEB APPLICATION SECURITY?	13
3.1 SECURITY IN THE WEB APPLICATION ARCHITECTURE.....	14
3.1.1 Application Layer.....	16
3.1.2 Application Program Interface	16
3.1.3 Middleware Layers	16
3.1.4 Infrastructure.....	16
3.2 SECURITY-AWARE DEVELOPMENT	17
3.2.1 Use a Security-Oriented Development Process and Methodology	17
3.2.1.1 SSE-CMM.....	19
3.2.2 Adopting an Effective Security Philosophy	20
3.2.3 Planning the Development Effort Realistically.....	20
3.2.4 Security Quality Assurance Throughout the Life Cycle.....	20
3.2.5 Accurate, Complete Specifications	20
3.2.6 Secure Design.....	21
3.2.6.1 Minimize Functionality.....	21
3.2.6.2 Minimize Component Size and Complexity	23
3.2.6.3 Minimize Trusted Components	23
3.2.6.4 Minimize Interfaces and Outputs.....	23
3.2.6.5 Avoid High-Risk Web Services, Protocols, and Components.....	24
3.2.6.6 Disable or Remove Unused Capabilities and Resources	24
3.2.6.7 Separate Data and Control.....	24
3.2.6.8 Protect All Sensitive Transactions	24
3.2.6.9 Protect Sensitive Data at Rest.....	25
3.2.6.10 Include Trustworthy Authentication and Authorization	25
3.2.6.11 Always Assume the Operating Environment Is Hostile	25
3.2.6.12 Always Assume that Third-Party Software Is Hostile	25

3.2.6.13 Never Trust Users and Browsers.....	26
3.2.6.14 Require and Authorize No Privileged Users or User Processes.....	26
3.2.6.15 Do Not Rely on Security Through Obscurity.....	26
3.2.6.16 Be Accurate in Your Assumptions About the Underlying Platform.....	27
3.2.6.17 Make Security Mechanisms Easy to Configure and Use.....	27
3.2.6.18 Risk Analysis of Application Design.....	27
3.2.7 Application Middleware Frameworks.....	28
3.2.8 Restricting the Development Environment.....	28
3.2.9 Writing Elegant Software.....	28
3.2.9.1 Document First.....	29
3.2.9.2 Keep Code Simple, Small, and Easy to Follow.....	29
3.2.9.3 Isolate Security Functionality.....	29
3.2.9.4 Be Careful with Multitasking and Multithreading.....	30
3.2.9.5 Use Secure Data Types.....	30
3.2.9.6 Reuse Proven Secure Code.....	30
3.2.9.7 Use Secure Programming Languages and Development Tools.....	31
3.2.9.8 Call Safely to External Resources.....	31
3.2.9.9 Use Escape Codes with Extreme Caution.....	33
3.2.9.10 Maintain a Consistent Coding Style.....	33
3.2.9.11 Find and Remove Bugs.....	34
3.2.9.12 Write for Reuse.....	34
3.2.9.13 Keep Third-Party Component Fixes and Security Patches Up to Date.....	34
3.2.9.14 Common Logic Errors to Avoid.....	35
3.2.10 Security-Aware Testing.....	36
3.2.10.1 Rules of Thumb for Security-Aware Testing.....	36
3.2.10.2 Code Reviews.....	37
3.2.10.3 Source Code Security Audits.....	38
3.2.10.4 Penetration Testing During Development.....	38
4.0 IMPLEMENTING SECURITY MECHANISMS.....	40
4.1 PUBLIC KEY-ENABLING.....	40
4.1.1 PK-Enabling: A Definition.....	41
4.1.2 Why PK-Enable?.....	42
4.1.3 When to PK-Enable.....	42
4.1.4 PK-Enabling Web Applications.....	44
4.1.4.1 Choosing an SSL 3.0 Tolerant Web Server.....	45
4.1.5 PK-Enabling Backend Applications: PKI Toolkits.....	46
4.2 IDENTIFICATION AND AUTHENTICATION MECHANISMS.....	46
4.2.1 Notification of Authentication.....	48
4.2.2 Client (Browser)-to-Server Trusted Path.....	49
4.2.2.1 Extending the Chain of Trust to a Backend Server.....	50
4.2.3 PKI-Based I&A.....	51
4.2.3.1 Browser Use of Hardware Tokens.....	51

4.2.4 Reusable (Static) Password I&A.....	51
4.2.4.1 Implementing Reusable Password I&A in Web Applications	52
4.2.4.2 Confidentiality and Integrity of Usernames and Passwords.....	53
4.2.4.3 Unsuccessful Log-In Attempts.....	54
4.2.4.4 Explicit Log-Out	54
4.2.4.5 Password Management	54
4.2.5 Single Sign-On Systems	55
4.2.5.1 Security Service APIs	56
4.2.5.2 SSPI in Windows NT and Windows 2000	57
4.2.5.3 Vulnerabilities of SSO Systems.....	57
4.2.6 Other I&A Technologies	58
4.2.6.1 One-Time (Dynamic) Password Systems	58
4.2.6.2 Biometric Authentication Systems	58
4.2.7 Pluggable Authentication Modules	59
4.3 AUTHORIZATION AND ACCESS CONTROL MECHANISMS	60
4.3.1 Implementing a Single Application Entry Point.....	60
4.3.1.1 Implementing the Web Portal's Checkpoint Program.....	61
4.3.1.2 Limitations of and Alternatives to Web Portal Security	62
4.3.1.3 Distributed Access Management Systems	64
4.3.2 Interoperation with System-Level Access Controls	64
4.3.2.1 Web Server Access Controls	64
4.3.2.2 Database Management System Access Controls	65
4.3.3 Least Privilege for Application Processes.....	66
4.3.3.1 Separation of Duties.....	66
4.3.3.2 Separation of Roles and Separation of Privileges.....	66
4.3.3.3 Minimizing Resources Available to Application Processes	66
4.3.3.4 Separation of Domains (Compartmentalization)	67
4.3.3.5 Least Privilege in Web Applications	67
4.3.3.6 Least Privilege in Database Applications	68
4.3.4 Role-Based Access Control.....	68
4.3.5 Additional Web Content Access Control Measures	69
4.3.5.1 Inhibit Copying of HTML Source Code.....	69
4.3.5.2 Inhibit Cutting and Pasting of Text Content.....	69
4.3.5.3 Inhibiting Screen Capture	70
4.3.6 Labeling and Marking of Output (Displayed and Printed)	70
4.3.6.1 Platform for Internet Content Selection Labels	70
4.3.7 Encryption of Data at Rest to Augment Access Controls	70
4.3.8 Session Control.....	71
4.3.8.1 Session Management Schemes	71
4.3.8.2 Session Time-Out	71
4.4 IMPLEMENTING CONFIDENTIALITY.....	72
4.4.1 Application Support for Object Reuse	72

4.4.1.1 Object Reuse in Java Applications	72
4.4.1.2 Avoiding Inadvertent Copies of Sensitive Data.....	72
4.4.1.3 Preventing Core Dumps of Sensitive Data.....	72
4.4.2 Confidentiality of Configuration Data and <i>include</i> Files.....	73
4.4.3 Confidentiality of User Identities	73
4.4.3.1 Do Not Hard Code User Credentials	73
4.4.3.2 Pass Sensitive Data Using POST, not GET.....	74
4.4.3.3 Exclude Confidential Data from Redirects	74
4.4.4 Validate URL Pathname Extensions	75
4.4.5 Limit Data Returned to Users	76
4.4.6 Do Not Trust Browsers to Store Sensitive Data.....	76
4.4.7 Application Integration with Data Encryption Mechanisms	76
4.4.7.1 Encryption Before Transmission.....	77
4.4.7.2 Encryption of Data at Rest.....	77
4.5 IMPLEMENTING DATA AND CODE INTEGRITY MECHANISMS	77
4.5.1 Implementing Hash.....	78
4.5.1.1 Hashing Data to Prevent Tampering.....	78
4.5.2 Integration of Digital Signature Mechanisms	79
4.5.2.1 Interface from Application to Digital Signature Mechanism.....	79
4.5.2.2 Digital Signature and Validation Capabilities for Browsers	80
4.5.2.3 Digital Signature Validation by Server Applications	80
4.5.2.4 Protection of Cryptographic Material Used for Digital Signature	80
4.5.2.5 Preventing Web Page Defacement.....	81
4.5.3 Code Integrity.....	81
4.5.3.1 Digital Signature of Mobile Code.....	82
4.6 ACCOUNTABILITY OF APPLICATION USERS	82
4.6.1 Application Integration with System Audit Log or Audit Middleware.....	83
4.6.1.1 Minimum Requirements for Application-Level Audit.....	83
4.6.1.2 Application Events to Be Audited.....	84
4.6.1.3 Application Logging if the Underlying Audit System Becomes Unavailable	84
4.6.1.4 Protection of Application Log Data Before It Reaches External Audit System.....	84
4.6.2 Application Security Violation Notifications.....	84
4.6.2.1 Application Integration with Intrusion/Violation Detection.....	84
4.7 NONREPUDIATION BY APPLICATION USERS	85
5.0 MAKING APPLICATIONS RESISTANT TO COMPROMISE AND DENIAL OF SERVICE	86
5.1 AVOIDING BUFFER OVERFLOW.....	86
5.2 AVOIDING CROSS-SITE SCRIPTING.....	87
5.3 INPUT VALIDATION	88
5.3.1 Designing Applications to Make Input Validation Easier	89

5.3.1.1	Clearly Define Acceptable Input Characteristics.....	90
5.3.1.2	Use Only Functions That Perform Bounds Checking.....	90
5.3.1.3	Pass Arguments in Environment Parameters.....	90
5.3.1.4	Suspend Processing Until Input Is Validated.....	90
5.3.1.5	Do Not Invoke Untrusted Programs from Trusted Programs.....	90
5.3.1.6	Use Only Independently Certified Third-Party Components.....	90
5.3.1.7	Validate Data Before Copying to a Database.....	90
5.3.1.8	Write Scripts to Check All Arguments.....	91
5.3.1.9	Protect Cookies at Rest and in Transit.....	91
5.3.1.10	Do Not Use Hidden Fields for User Input.....	91
5.3.2	Rejection and Sanitization of Bad Input.....	92
5.3.3	Notification of Correct Input.....	92
5.3.4	Validations to Perform.....	93
5.3.4.1	Type Checks.....	93
5.3.4.2	Format and Syntax Checks.....	93
5.3.4.3	Parameter and Character Validity Checks.....	93
5.3.4.4	Divide-by-Zero Checks.....	99
5.3.4.5	Check for User Input to Formatting Routines.....	99
5.3.4.6	Check for Session Token to Prevent URL Manipulation.....	100
5.3.4.7	HTTP Header Checks.....	100
5.3.5	Virus Scanning.....	101
5.4	INTEGRITY AND INPUT VALIDATION IN DATABASE APPLICATIONS.....	102
5.4.1.1	Reparsing Requests and Data for Backend Databases.....	102
5.4.1.2	Avoiding Direct SQL Injection.....	102
5.4.2	Validate Originators of Data and HTML.....	104
5.5	APPLICATION AWARENESS OF THE OPERATING ENVIRONMENT.....	105
5.6	PROTECTING APPLICATION CONFIGURATION DATA.....	105
5.7	INTERPROCESS AUTHENTICATION: BEYOND CHALLENGE AND RESPONSE	105
	105
5.7.1	Kerberos.....	105
5.7.2	X.509.....	106
5.7.3	Secure Remote Procedure Call.....	106
5.8	USE OF MOBILE CODE.....	106
5.8.1	Use Only Approved Mobile Code Technologies.....	106
5.8.2	Mobile Code Signature and Validation.....	107
5.8.3	Secure Distribution of Mobile Code.....	107
6.0	MAKING APPLICATIONS RESISTANT TO INTERNAL FAILURE.....	109
6.1	CONTROLLING OPERATION AND AVAILABILITY.....	109
6.1.1	Availability Requirements for DoD Applications.....	109
6.1.2	Input Time-Outs and Load Level Limits.....	111

6.1.3 Adjusting to Unresponsive Output	111
6.1.4 Preventing Race Conditions	111
6.1.4.1 What is a Race Condition?	111
6.1.4.2 Preventing Deadlocks.....	112
6.1.4.3 Preventing Sequence Conditions	114
6.1.5 Application Invocation of Backup.....	115
6.2 ERROR AND EXCEPTION HANDLING AND RECOVERY.....	115
6.2.1 Failing Safe	115
6.2.2 Error Detection.....	116
6.2.3 Resistance to Denial of Service.....	116
6.2.4 Administrator-Configurable Error Responses	116
6.2.5 Transaction Rollback and Checkpoint Restart.....	116
6.2.6 Consistency Checking Before Restart.....	117
6.2.7 Safe Error Messages.....	117
6.2.8 Error Logging.....	118
7.0 DEVELOPMENT TOOLS.....	119
7.1 APPLICATION MIDDLEWARE FRAMEWORKS.....	119
7.1.1 Distributed Computing Environment.....	119
7.1.2 .NET and Distributed Component Object Model.....	119
7.1.3 Common Object Request Broker Architecture.....	120
7.1.4 Simple Object Access Protocol.....	121
7.2 OTHER DEVELOPMENT TOOLS.....	122
7.2.1 Compilers and Linkers	122
7.2.2 Debuggers	122
7.2.3 Web Authoring Tools.....	123
7.2.3.1 WYSIWYG Tools versus. Text Editors and HTML Editors.....	123
7.3 PROGRAMMING LANGUAGE SECURITY.....	125
8.0 PREPARING APPLICATIONS FOR DEPLOYMENT	126
8.1 PREPARING CODE FOR DEPLOYMENT.....	126
8.1.1 Remove Debugger Hooks and Other Developer Backdoors	126
8.1.1.1 Explicit Debugger Commands.....	126
8.1.1.2 Implicit Debugger Commands.....	126
8.1.2 Remove Data-Collecting Trapdoors	127
8.1.3 Remove Hard-Coded Credentials	128
8.1.4 Remove Default Accounts	128
8.1.5 Replace Relative Pathnames	129
8.1.6 Remove Sensitive Comments.....	129
8.1.7 Remove Unnecessary Files, Pathnames, and URLs	130
8.1.8 Remove Unneeded Calls	130
8.2 RUN-TIME CONSIDERATIONS.....	131

8.2.1 Load Initialization Values Safely.....	131
8.3 SECURE INSTALLATION AND CONFIGURATION	131
8.3.1 Configure Safely and Use Safe Defaults	131
APPENDIX A: ABBREVIATIONS AND ACRONYMS.....	132
APPENDIX B: REFERENCES AND SUGGESTED READING.....	137
B.1 REFERENCES USED TO PREPARE THIS DOCUMENT.....	137
B.2 SUGGESTED FURTHER READING	140
B.3 BOOKS	152
APPENDIX C: THIRD-PARTY SECURITY TOOLS	154
C.1 SELECTING THIRD-PARTY TOOLS.....	154
C.2 THE TOOLS.....	158
APPENDIX D: PROGRAMMING LANGUAGE SECURITY.....	172
D.1 C AND C++.....	172
D.2 VISUAL BASIC	178
D.3 JAVA	178
D.4 HYPERTEXT MARKUP LANGUAGE.....	184
D.5 XML AND SDML.....	185
D.6 ASP AND JSP.....	186
D.7 CGI AND PERL	197
D.8 STRUCTURED QUERY LANGUAGE.....	201
D.9 SHELL SCRIPTING LANGUAGES.....	202
D.10 TOOL COMMAND LANGUAGE.....	202
D.11 PHP	202
D.12 PYTHON	206
APPENDIX E: SECURITY-ENHANCING LEGACY APPLICATIONS	207
E.1 WEB ENABLING FOR SECURITY.....	207
E.2 APPLICATION FIREWALLS	210
E.3 SECURITY WRAPPERS.....	210

1.0 INTRODUCTION

1.1 PURPOSE

This document provides guidance and recommendations to developers interested in securing their applications. The document builds on the application security requirements defined in the *Recommended Standard Application Security Requirements* document, drafted as a precursor to this guide, and provides developers with guidance on ways to implement those requirements. The guidance addresses applications of various types, including Web applications and Web applications that interoperate with backend databases and other legacy servers. The document discusses ways to avoid general vulnerabilities and common programming and coding errors. The document discusses development methodologies and techniques on ways to ensure that security mechanisms are implemented in applications early in the development life cycle.

It is anticipated that the guidance presented herein will be used as a developer's tool to design security into applications during the development phase. The guidance techniques and examples will aid application developers to eliminate potential application vulnerabilities and security flaws during the early phases of the application life cycle. At a minimum, the guidance should be used to provoke thought about application security within the minds of application designers, developers, and programmers.

It is assumed that the readers of this guide will already be well versed in effective software development and testing practices and will have a thorough understanding of, and experience with Web technologies and the development of Web applications. This document is not intended as a primer to teach novice developers how to develop Web applications. It is intended to assist experienced Web developers in meeting the more stringent and sometimes unique security requirements of Department of Defense (DoD) Web applications.

DoD programs should use this guidance as a resource to assist their developers when implementing security requirements in applications. The guidance found in the document includes a compilation of application development recommendations, references, and pointers to security toolkits, as well as code and best practices examples to implement security mechanisms in applications.

This document can be used as a reference during Phase I (Definition) and Phase II (Verification) of the DoD Information Technology Security Certification and Accreditation Process (DITSCAP) and specifically during Phase II system development activities. In addition, this guidance should be used in conjunction with Defense Information Systems Agency's (DISA) Security Technical Implementation Guides (STIGs) during the development cycle. The STIGs can be downloaded from the Information Assurance Support Environment (IASE) Web site (<http://iase.disa.mil>) or Field Security Operations (FSO) Guides (<http://guides.ritchie.disa.mil>) Web site. Developing on a STIG-compliant system and applying the recommended requirements listed in this document will help to ensure a high level of security within your application.

This developer's guide is the second document in a series of documents being prepared by the Applications and Computing Security Division, Center for Information Assurance Applications. Building on the application security requirements presented in the first document, this developer's guide will assist developers with implementation of those requirements and present guidance on ways to limit the general application vulnerabilities presented in *Recommended Standard Application Security Requirements*. The third and fourth documents in this series will identify application security assessment tools and present assessment methodologies that can be used to validate the application security mechanisms.

1.2 SCOPE

The primary focus of this document is the development of Web applications including Web applications that interoperate with backend databases and other legacy servers. The emphasis is on security of Web server application components; the rationale for this emphasis is that server applications are much more exposed than browsers, and therefore their vulnerabilities are exploitable by a much larger population of potential attackers.

Many of the security mechanisms and associated techniques described in this document are relevant only for applications that run on DoD private Web servers, whether classified or unclassified, mission critical or not. Because DoD publicly accessible (public) Web servers do not require user authentication, authorization, accountability, or nonrepudiation, much of the material in this document will not be relevant for developers of DoD public Web applications.

Nevertheless, the access control (including role-based access control), integrity, and availability mechanisms and techniques described should be equally relevant for developers of private and public Web applications. The DoD Public Key Infrastructure Program Management Office (PKI PMO) Frequently Asked Questions (FAQ – see <http://www.c3i.osd.mil/org/sio/ia/pki/faq.html>) provides specific definitions of private and public Web servers. Also of interest for developers of public Web applications may be the National Institutes of Standards and Technology (NIST) Special Publication 800-44, *Guidelines on Security Public Web Servers*, specifically Section 5 on security Web content and Section 6 on authentication and encryption technologies. See <http://csrc.nist.gov/publications/drafts/PP-SecuringWebServers-RFC.pdf>.

An attempt has been made to link specific guidance to some of the more common vulnerabilities, to help reduce the occurrence of these vulnerabilities in applications. The vulnerability list is not all-inclusive and will be amended as required. This document does contain vulnerability remediation information and specific guidance on methods to avoid some of the general and specific vulnerabilities.

This living document contains a general guidance and best practices on implementing security in applications. The guidance and coding examples will continue to be refined and enlarged as time goes on. Revisions and updates to this document are planned annually.

1.2.1 Subjects Not Addressed in This Document

The following subjects are considered outside the scope of the main body of this document, or they have been addressed in a limited way in the appendices:

- *Backend databases:* Security of backend databases is discussed only in terms of the security of the interface between the backend database and the Web application. Implementing database security and security in non-Web database applications is outside the scope of this document.
- *Legacy applications:* Discussion of retrofitting of security into legacy applications is discussed briefly in Appendix E and is limited to the implementation of security when Web enabling legacy applications.
- *Browser security:* The browser security guidance in the body of this document pertains to the secure interoperation of and interfaces between Web server applications and commodity desktop browsers. Browser security itself is the result of correct configuration by the installer and correct use by the user, rather than of any effort on the part of the application developer. It is hoped that you understand browser security capabilities and vulnerabilities well to better design the server application's security mechanisms and interfaces to make up for the security limitations of the browser. There are some useful references in Appendix B pertaining to browser security.
- *Mobile network-based Web applications:* Web applications implemented on handheld computers, mobile applications (including Web applications) on handheld computers interconnected by wireless networks, agent-based applications (often used in conjunction with active networks), and collaboration applications (often in multicast networks) will be addressed in a future version of the document.
- *E-mail and messaging applications:* Although it is recognized that e-mail and messaging applications may be incorporated into the larger Web application, security requirements and implementation guidance for DoD e-mail and messaging applications are considered outside the scope of this document. They are being addressed by other DoD initiatives, such as the Defense Message System and DoD PKI.

1.3 INTENDED AUDIENCE

Application developers should use this document as a guide for implementing security features in their applications so they function securely on DoD systems. The document will help application developers understand what needs to be secured so they can develop and insert specific application security controls and avoid creating vulnerabilities in their applications. Examples presented can be modified as required by developers for inclusion in their applications.

Some of the information in this document may be beneficial to system administrators and system security engineers. However, those personnel interested in configuration and operational security requirements should refer to the various STIGs currently available from DISA.

1.4 NOTE ON STYLE

For clarity and conciseness, in addressing this document has been written in the second person with the imperative mood. This may make the document's tone seem rather assertive at times; however, this was preferred rather than writing in the softer, less direct third person.

2.0 BACKGROUND

The mission of the DISA Application and Computing Security Division (Code API2) is to provide for the identification, development, systems engineering, prototyping, provisioning, and implementation of various technologies supporting the defense-in-depth (DID) concept for multi-layered protection of the global applications and computing infrastructure of the global information grid (GIG).

The DISA Application and Computing Security Division believes that a core set of application security requirements and common vulnerabilities for all applications exists. This document aids developers with the implementation of those requirements in their applications. The Application and Computing Security Division will use this document to compile and categorize developer guidance, helpful security tool kits, and coding best practices.

2.1 ORGANIZATION AND CONTENT OF THIS DOCUMENT

This document is organized as follows:

Executive Summary: Highlights the key topics addressed in this document.

Section 1, Introduction: Describes the purpose and scope of the developer guidance provided in the document, the type of problems to be solved, the audience targeted, and the prerequisite knowledge expected of the reader.

Section 2, Background: Describes the application security problems at a high level and discusses the role of the DISA Application Security organization, the purpose of the *Recommended Standard Application Security Requirements* document, and its relationship to this *Application Security Developer's Guide*. Also introduces plans for future documents in this series on guidance for application developer.

Section 3, Security-Aware Development: Provides guidance on software development practices and software engineering principles that promote the creation of secure applications.

Section 4, Adding Required Technical Security Mechanisms to Applications: Provides guidance to help developers implement the technical security mechanisms specified in the Recommended Standard Application Security Requirements within applications, or to use the appropriate application program interfaces (APIs) to integrate applications with external security mechanisms. These technical security mechanisms are designed primarily to protect the data that applications process; they include mechanisms for identification and user identification and authentication (I&A), access control, data integrity and availability, and accountability and nonrepudiation of actions taken by users of the application.

Section 5, Making Applications Resistant to Compromise: Provides guidance beyond the implementation of technical security mechanisms, on designing and implementing hacker-proof applications. This guidance includes tips on minimizing the likelihood of exploitable vulnerabilities, safe

design and coding practices, implementation of interprocess trust mechanisms and effective input validations, and recommendations of add-on technical mechanisms that can help protect the integrity and availability of the application program itself, versus the data it handles.

Section 6, Making Applications Resistant to Internal Failure: Provides guidance beyond making the application resistant to external attack, on designing and implementing correctly operating, fail-safe applications. This guidance is intended to further improve application availability by minimizing design and coding errors. It includes tips on avoiding problematic constructs and implementing effective error handling to minimize the frequency and impact of application failures.

Section 7, Choosing and Using Development Tools to Promote Security: Provides guidance on the selection of development tools, such as compilers, linkers, debuggers, and libraries—that promote rather than impede application security, and the security-aware use of those tools. This guidance includes discussions of common security violations within various development tools and provides tips on how to work around or fix these problems.

Section 8, Preparing Applications for Deployment: Provides guidance on cleaning up application code to remove any residual security problems before deployment.

Appendix A, Abbreviations and Acronyms: Defines the acronyms and abbreviations used in this document.

Appendix B, References and Suggested Reading: Presents a list of printed and on-line documents and books used when preparing this document and lists additional documentary resources recommended for use by application developers.

Appendix C, Third-Party Security Tools: Provides guidance on selecting commercial off-the-shelf (COTS) applications and application security components. It also presents a list of available government off-the-shelf (GOTS), COTS, and (limited) open source application security components and development tools that developers may find useful in implementing secure applications. Cross-references to Appendix C appear throughout Sections 3 and 4 of this document, introducing components and tools that may help the developer enact particular recommendations.

Appendix D, Programming Language Security: Discusses security issues pertaining to specific programming languages used in Web development.

Appendix E, Security-Enhancing Legacy Applications: Provides guidance on retrofitting existing applications by adding security mechanisms without rewriting the application code. Special emphasis is given to the security issues associated with Web-enabling of legacy applications.

2.2 HOW TO USE THIS DOCUMENT

The guidance in this document is not intended to define a secure application development methodology. It is, however, intended to define a set of secure application development strategies.

Accordingly, the recommended activities and techniques in Sections 3 through 8 should be integrated into the Web application development process and methodology you use to help improve the security of your software development process, and of the Web applications that are produced by that process.

In practical terms, this document is meant to provide guidance that will enable the Web application developer to satisfy the DISA *Recommended Standard Application Security Requirements*. To this end, this document provides developer guidance on how to

1. Implement security mechanisms and capabilities specified in the DISA *Recommended Standard Application Security Requirements* within Web applications; these are the mechanisms and capabilities that enable Web applications to protect the information they process
2. Ensure that applications can protect themselves against compromise and denial of service-induced failure. This guidance is intended to satisfy the DISA *Recommended Standard Application Security Requirements* pertaining to application integrity and availability, and moreover to address the 22 categories of common application vulnerabilities defined in that document.

Tables 2-1 and 2-2 are provided as quick cross-references for developers. Table 2-1 is a cross-reference matrix that maps the individual security requirements in Section 4 of *Recommended Standard Application Security Requirements* to the sections and subsections the present document. Table 2-2 provides a cross-reference matrix that maps the 22 common vulnerabilities identified in Section 3 of *Recommended Standard Application Security Requirements* with the relevant sections and subsections of the present document.

If desired, these cross-reference matrices can be used in conjunction with or in lieu of the table contents page to navigate through this document to identify the sections of the document that will address specific requirements defined in the DISA requirements document. In addition, they can be used during the assessment of the Web application specification, design, and implementation, to ensure that each of these satisfies the necessary requirements.

Table 2-1: Security Service Requirements and Associated Developer Guidance

Requirements Document Section No.	Developer Guidance Section No.
Requirements for All IA Mechanisms	
4.0.1: No bypass of security controls	4.3.1, 4.3.2.1
4.0.2: Integrity of external security	3.2.6.6, 3.2.6.11-13, 3.2.6.16, 3.2.9.8, 3.2.9.13
4.0.3: Integrity of external operation	3.2.6.6, 3.2.6.11-13, 3.2.6.16, 3.2.9.8, 3.2.9.13
4.0.4: Integrity of platform security	3.2.6.16
4.0.5: Integrity of platform operation and data	3.2.6.16
4.0.6: Interoperability with DoD PKI	4.1
4.0.7: Class 4 certificates	4.2
4.0.8: PKE of applications	4.1
4.0.9: Approval of crypto	4.1
4.0.10 High-risk services	3.2.6.5

Requirements Document Section No.	Developer Guidance Section No.
4.0.11: Application deployment	8.0
4.0.12: Privileged processes	3.2.6.3, 3.2.6.14, 4.3.3, G1.3.1
4.0.13: HTML comments	7.2.3.1.1, 8.1.1.6
4.0.14: Browser application facilities	Appendix F
General Application Identification and Authentication	
4.1.1: Authentication of users	3.2.6.10, 4.2
4.1.2: Required I&A technology	4.2
4.1.3: Desirable I&A technology	4.2
4.1.4: Authentication chain of trust	4.2.2.1
4.1.5: I&A trusted path	4.2.2
4.1.6: Backend system I&A	4.2.2.1
4.1.7: Maximum number of unsuccessful attempts	4.2.4.3
4.1.8: I&A lockout period	4.2.4.4
4.1.9: I&A using PKI certificates	4.2.2.1, Appendix C
4.1.10: I&A using PKI tokens	4.2.2, 4.2.3.1, Appendix C
4.1.11: Private Web server I&A	4.1.3
4.1.12: Public Web server I&A	4.1.3
4.1.13: Classified Web server I&A	4.1.3
4.1.14: Browser support for tokens	4.2.3.1, Appendix C
4.1.15: No I&A by Java applets	4.2
4.1.16: Support for Class 4 certs	4.2
4.1.17: Support for CAC	4.2.3.1
4.1.18: I&A using biometrics	4.2.6.2, Appendix C
4.1.19: Strong passwords	4.2.4.5
4.1.20: Password changes	4.2.4.5
4.1.21: Password expiration	4.2.4.5
4.1.22: Selection of new password	4.2.4.5
4.1.23: Group I&A	4.3.1.1
4.1.24: Confidentiality of transmitted passwords	3.2.6.8, 4.2.3., 4.4.3.2
4.1.25: Confidentiality of password during reformatting	Determined not to be an application function.
4.1.26: Integrity of I&A data	4.2.3
4.1.27: I&A between client and server processes	4.2, 4.2.2, 5.7.1, 5.7.2, 5.7.3
4.1.28: Interprocess I&A in peer-to-peer applications	5.7.1, 5.7.2, 5.7.3
4.1.29: Warning message to authenticated user	4.2.1, 4.3.1.2, 4.3.6.1
4.1.30: Unique usernames	4.2.3.1, 4.2.4.1.1, 4.2.4.2
4.1.31: Unique passwords	4.2.3.1, 4.2.4.1.1, 4.2.4.2
4.1.32: No anonymous accounts	4.2.2, 4.2.4.5, 4.3.3.5
4.1.33: Authentication requires trustworthy credential	4.2
4.1.34: Freedom in assigning usernames and group IDs	4.2.4.5
General Application Authorization and Access Control	
4.2.1: Authorization	4.3
4.2.2: Authorization information management	4.3

Requirements Document Section No.	Developer Guidance Section No.
4.2.3: Authorization information confidentiality	4.3
4.2.4: Authorization information integrity	4.3
4.2.5: Authorization information availability	4.3
4.2.6: Interprocess authorization	5.2, 5.7.1, 5.7.2, 5.7.3, 4.2.2.1
4.2.7: RBAC for privileged accounts	4.3.4, Appendix C
4.2.8: RBAC in classified applications	4.3.4
4.2.9: Maximum number of sessions	4.3.8.1
4.2.10: Inactivity time-out	4.3.8.2
4.2.11: Access control for classified data	4.3
4.2.12: Access control for sensitive and Mission Category I unclassified data	4.3
4.2.13: Data change notification	4.3.2.2.1
4.2.14: Labeling of classified data	4.3.6
4.2.15: Labeling of unclassified data	4.3.6
4.2.16: Marking of output	4.3.6
4.2.17: Invalid pathname references	4.3.2.1.1, 8.1.1.7
4.2.18: Truncated pathnames	3.2.9.8.5, 4.3.2.1.2
4.2.19: Relative pathnames	3.2.9.8.5, 4.3.2.1.2, 7.2.3.1.1.2, 8.1.1.5
4.2.20: Relative pathnames input by users	4.3.2.1.2
4.2.21: Rejection of directly entered URLs	4.3.1.2
4.2.22: Browser protection of user identity	4.4.3, Appendix F
4.2.23: CGI scripts	4.3.3.5, G1.7
General Application Confidentiality	
4.3.1: Encryption API	4.4.7
4.3.2: Nondisclosure of cleartext data	4.2.4.1, 4.4.7.1
4.3.3: Encryption before transmission	3.2.6.8, 4.2.3.1, 4.4.3.2, 4.4.7.1
4.3.4: Encryption of stored data	3.2.6.9, 4.4.7.2, Appendix C
4.3.5: Protection of cryptokeys	4.4.7
4.3.6: PKI encryption certificates	4.2
4.3.7: Application object reuse	4.4.1
4.3.8: Confidentiality of crypto material	3.2.6.1-9, 4.4.7
4.3.9: Confidentiality of user identities	4.4.3, 8.1.1.3
General Application Integrity	
4.4.1: Integrity of transmitted data	3.2.6.1-18, 4.5.1, 4.5.3
4.4.2: Integrity of transmitted application code	4.5.3
4.4.3: Integrity of stored data	3.2.6.9
4.4.4: Integrity mechanism validation	4.5.3, Appendix C
4.4.5: Validation of parameters	3.2.9.8.3, 4.5, 5.3.4.3
4.4.6: Notification of acceptable input	5.3.3
4.4.7: Validation of user input	5.3, G1.6.2.2
4.4.8: Rejection of incorrect input	5.3.2
4.4.9: Input validations by server	5.3
4.4.10: Data containing active content	5.3.4.1
4.4.11: Application process integrity	4.3, 5.0
4.4.12: Integrity of transmitted application code	4.5.3
4.4.13: Application configuration integrity	5.6
4.4.14: Application executable integrity	4.5.3
4.4.15: Time and date stamp of data	4.3.2.2.1

Requirements Document Section No.	Developer Guidance Section No.
modification	
4.4.16: Display of data time and date stamp	4.3.2.2.1
4.4.17: Resolution of mode changes	Determined not to be relevant
4.4.18: Initialization of variables	8.1.2.1
4.4.19: Integrity of crypto data	3.2.6.9, 4.4.7, 4.5.3
4.4.20: Cryptokey revocation	4.4.7
4.4.21: Certificate revocation	4.1.4, 4.4.7
4.4.22: Signature of code	4.5, 4.5.2.1-5
4.4.23: Use of hidden fields	4.2.2, 4.5.1, 5.3.1.10
General Application Availability	
4.5.1: Data availability	6.1.1.4-5
4.5.2: Server application availability	6.1.1
4.5.3: Mission Category 1 client application availability	6.1.1
4.5.4: Maintenance of secure state	6.2.1
4.5.5: Application failure notification	6.2.2, 6.2.4
4.5.6: Secure application recovery	6.2.1
4.5.7: Application denial of service	6.0, G1.1.2
4.5.8: Error handling and recovery	6.2.4-8
4.5.9: Missing files	6.2.1
4.5.10: Key recovery	4.4.7
General Application Accountability	
4.6.1: Audit and event logging mechanism	4.6.1.2, Appendix C
4.6.2: Configurable audit and log parameters	4.6.1.1, 4.6.1.2
4.6.3: Events to be audited and logged	4.6.1.2
4.6.4: Binding of user ID to audit record	4.6.1.2
4.6.5: Audit information captured by classified applications	4.6.1.2
4.6.6: Audit information captured by sensitive and nonpublic access applications	4.6.1.2
4.6.7: Audit information captured by public access applications	4.6.1.2
4.6.8: Protection of audit records	4.6.1.4
4.6.9: Audit trail fill thresholds	4.6.1.1
4.6.10: Audit failure	4.6.1.1, 4.6.1.3
4.6.11: Security violation notifications	4.6.2.1
4.6.12: Audit trail viewing and reporting tool	4.6.1.1
General Application Nonrepudiation	
4.7.1: Digital signature of created and transmitted data	4.5.2.1- 4.5.2.5.1
4.7.2: Digital signature of received data (proof of delivery)	4.5.2.2
4.7.3: Digital signature validations	4.5.2.3, Appendix C
4.7.4: Protection of digital signature security data	4.5.2.4
4.7.5: PKE of e-mail applications	1.2 Note
Use of Mobile Code in Applications	
4.8.1: Category 1 and 2 mobile code source	4.5.3.1, 5.8
4.8.2: Category 1 and 2 mobile code execution	5.8

Requirements Document Section No.	Developer Guidance Section No.
4.8.3: Category 2 mobile code execution	5.8
4.8.4: Category 2 mobile code notification	5.8
4.8.5: Category 3 mobile code	5.8.1
4.8.6: Emerging mobile code technology	5.8
4.8.7: Mobile code in e-mail messages	1.2 Note
4.8.8: E-mail client mobile code notification	1.2 Note

Table 2-2: Common Vulnerabilities and Associated Developer Guidance

Vulnerability Code and Description	Guidance Section No.
V1: Inadequate I&A	4.2
V2: Insufficient access control	4.3
V3: Improper integration of application components	8.1.3.1
V4: Weak passwords	4.2.4.5
V5: Plain text communication of sensitive information	3.2.6.8, 4.4.3.2
V6: Incorrect reparsing of data	4.5.2.9.1
V7: Buffer overflow	5.1, G1.1.2
V8: Lack of adequate parameter validation	3.2.9.8.3, 4.5.2.4, 5.3.4.3
V9: Input validation of data containing active content	5.3.4.1
V10: Acceptance meta code embedded within input data	3.2.9.8.4, 5.3.4.1, 5.3.4.3.2.1-2
V11: Acceptance of illegal characters in SQL queries	5.4.1.2
V12: Use of relative pathnames	4.3.2.1.2, 8.1.1.5
V13: Acceptance of truncated pathnames	4.3.2.1.2
V14: Links to pathnames no longer present on the server	4.3.2.1.1, 8.1.1.7
V15: Inefficient error handling and error recovery	6.2
V16: CGI script holes	G1.7
V17: Presence of developer backdoors	7.2.2, 8.1.1.1
V18: Password grabbing and replay	4.2.4.1
V19: Susceptibility of cookies to content changes	4.2.4.1
V20: Lack of access controls on directly typed URLs	4.3.1.2.1
V21: Use of hidden fields	5.3.1.10
V22: Web page defacement	4.3.5

In addition to the guidance in the body of this document, there are several appendices. Of particular interest is Appendix B, which provides not only a list of the references used to develop this document, but an extensive list of suggested further reading that includes both links to on-line documentation and references to books. We consider that reading to be potentially helpful in further elucidating ideas

presented in this document, going into much greater detail in some areas, and also introducing interesting approaches other developers have used to solve specific problems in securing their Web applications.

Appendix C provides an extensive list of third-party products and tools that developers may find helpful when seeking existing security mechanisms and functionality that they can integrate into their applications, and when seeking development tools to support the secure development methodology and process.

Moreover, Appendix C provides guidance for developers and systems engineers in some criteria that third-party products should satisfy to qualify for use in DoD Web applications. Appendix C is not intended to replace or even formally augment DoD procurement policy; however, it is intended to provide a kind of first cut set of minimal requirements that developers and engineers should keep in mind when assessing a particular third-party candidate product, thus increasing the likelihood that the products the developer/engineer does recommend for procurement will satisfy the formal procurement requirements.

Appendix D of this document provides guidance specific to the use of individual programming languages. This guidance is collected in Appendix D, rather than included in the body of this document, in recognition of the fact that most developers will not be interested in the security issues of programming languages they have no intention of using. Appendix D is intended to augment and specify the more general guidance given in the body of this document in areas such as input validation, buffer overflow, and others.

Appendix E of this document provides guidance to developers who are adding security to legacy applications by Web enabling those applications.

3.0 WHAT IS WEB APPLICATION SECURITY?

Web application security has two main objectives:

1. Providing the security functionality to or within the application that it needs to protect the data it processes
2. Protecting the application itself from compromise.

Both of these objectives are expressed in DISA's *Recommended Application Security Requirements* (Version 1.1, 6 June 2002), and the guidance in this document is specifically intended to help you achieve these objectives in the applications you develop.

Objective #1 is addressed mainly in Section 4, which discusses how to implement the security mechanisms to be used by applications to protect the data they process. In the case of DOD Web applications, many of the security mechanisms will involve the technologies provided by the DoD PKI. However, not all security needs can be met by DoD PKI, so this document is more than a primer on public key-enabling (PKE), and indeed refers developers to the already significant body of documentation produced by DISA on PKE of DoD applications. Objective #1 will mostly concern security services provided to the application (by middleware or operating system level mechanisms), rather than functions coded within the application. Your task as a developer will be implementing the application's interfaces to such services correctly and consistently.

Objective #2 is addressed mainly in Section 5 and 6, which discuss how to make the application resistant to compromise and resistant to internal errors that could make the application susceptible to denial-of-service attacks. For Objective #2, it may help to understand why hackers who have traditionally targeted networks and operating systems have begun increasingly to attack applications.

In general, hackers attack applications in an attempt to change their privilege levels so they can

- Do more than they are supposed to do
- See more than they are supposed to see
- Deny service to authorized users.

There are actually only a few, quite popular attack styles that account for the vast majority of known attacks. These are

- Overflows
- String format attacks
- Input validation errors
- Race conditions

These attack styles are launched against all types of objects, including network devices, operating systems, middleware, and applications. In fact, all software is vulnerable to these attacks, no matter where that software resides in the application architecture. But whereas the security protection of

operating systems and networks is a well-established discipline, security protection of applications is a fairly new science. Hackers are turning increasingly to applications as targets because networks and operating systems are growing more difficult to crack. Applications are the new frontier for hackers—and must also become the new frontier for security. Unlike application security mechanisms, the techniques the application uses to protect its own integrity and availability (Objective #2) may well require development of logic within the application itself. Because such kinds of attacks may be relatively unfamiliar to many developers, a significant amount of information has been provided in this document on the countermeasures—the most significant data input being validation and error handling and recovery—you can implement within the application itself to minimize its vulnerability to these attacks.

First, however, it may be useful to understand what we mean when we refer to a Web application, as well as its relationship to security middleware and the underlying security mechanisms in the operating system and throughout the operating environment.

3.1 SECURITY IN THE WEB APPLICATION ARCHITECTURE

Figure 3-1 depicts a notional architecture for a Web server application (see Appendix A for abbreviations and acronyms).

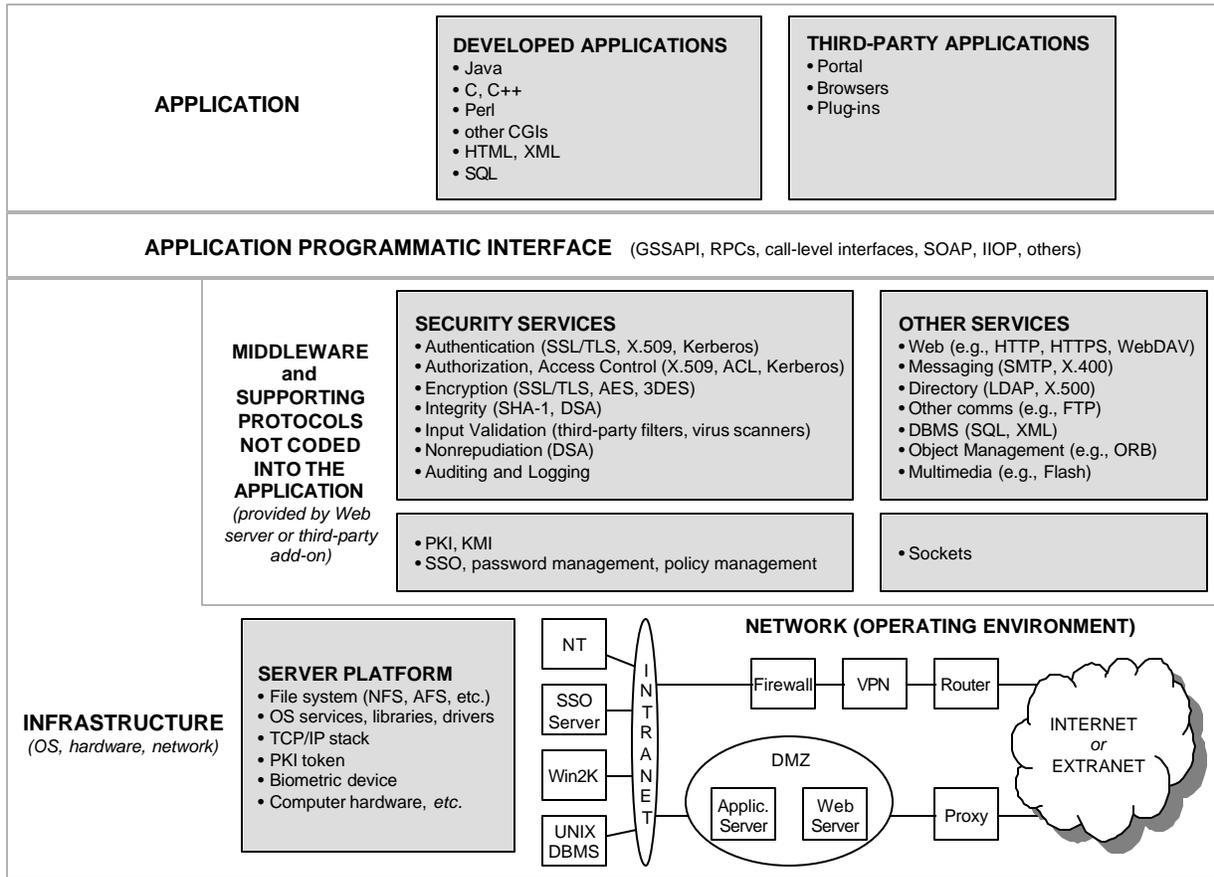


Figure 1: Notional Web Application Architecture

The upper layers of the application—the application programs and APIs, and the middleware layers below it—are logical representations. The lower infrastructure layer depicts both a logical representation of the underlying server platform infrastructure and a physical representation of the operating environment that also provides services indirectly to the Web application (e.g., single sign-on [SSO], firewall, virtual private network [VPN], proxy servers). Depending on who its user community is, the application itself may run on either of the servers in the demilitarized zone (DMZ), or possibly on one of the servers on the intranet.

The upper layer of the notional architecture reflects developed applications. To some extent, the guidance provided in this document can form the basis for defining criteria for evaluating third-party applications, particularly those that will be integrated with custom-developed programs to create the larger Web application. The application programmatic interfaces, middleware, and infrastructure are of interest. They are the layers that will enable the applications to obtain many of the security services and protections they need from preexisting sources like the DoD PKI, the Web server, and the underlying operating system, instead of having to include them all in their own program code.

3.1.1 Application Layer

This is where the application-unique functionality—that is, the application programs—is implemented. The application may be custom developed, or it may be a third-party (commercial or public domain) program or it may be integrated to incorporate both third-party and custom-developed functionality; for example, a commercial Web portal application through which are integrated several custom-developed HTML database forms and Java servlets.

3.1.2 Application Program Interface

The APIs provide the application program's interfaces to the underlying services provided by middleware or by infrastructure. They may include standards-based APIs, such the Generic Security Service API (GSS-API) defined by the Internet Engineering Task Force (IETF) Requests for Comments (RFCs) RFC 1508 and RFC 1509, the Simple Object Access Protocol (SOAP) defined by the World Wide Web Consortium (WC3; for more on SOAP, see Section 7.1.4), or SAML (Security Assertion Markup Language, an XML-based security standard for exchanging authentication and authorization information by the Organization for the Advancement of Structured Information Standards [OASIS]; see Appendix D, Section D.5 for more information on XML security standards). Or they may be proprietary APIs defined by the supplier of the middleware. Even more, they may be simple call-level interfaces such as a Remote Procedure Call (RPC; see Section 5.7.3 for information on Secure RPC).

3.1.3 Middleware Layers

Security middleware is designed to create self-contained security service functionality that can be used by multiple applications. On the client, this middleware takes the form of secure sockets layer (SSL) and any other PKI services (e.g., digital signature) linked into the browser. On the server, security middleware can be used to implement all of the standard security services the server application will need, including authentication, authorization, access control (beyond what is provided by the operating system), encryption, integrity mechanisms, auditing, nonrepudiation, and even some input validations and filtering.

Security middleware also incorporates the application-layer portions of security infrastructure services such as PKI, SSO, and biometric systems relied upon by the application. These lower-level middleware services may interface directly with the application, or they may interface with higher-level middleware services that use them to provide their own services to the application. For example, SSL may rely on the underlying PKI to provide it access to the cryptokeys and X.509 certificates it needs to provide the application's authentication and encryption services.

3.1.4 Infrastructure

Broadly, the infrastructure is the operating environment in which the application runs. This includes the platform environment—the underlying operating system services and facilities, networking services, and hardware—the computing platform itself, and any cryptographic token, biometric device, and the like

with which the application interoperates. The infrastructure can also be seen in a wider context, as the entire networked environment in which the application, on its platform infrastructure, operates. Some of the security and other services used by the application may be provided to it over the network by other physical systems, such as SSO servers, backend databases, directories, and message-handling systems.

Other systems in the network infrastructure provide services to the application indirectly. By performing certain security functions—such as VPN, proxy and firewall protection—they create another layer of defenses that envelop the application and its underlying middleware, operating system, and hardware platform, which collectively provide the security functionality of and direct security protection to the application itself. For example, an application environment (network) that is carefully isolated from outside connections by use of a strong firewall, and in which all client-to-server and server-to-server network connections are encrypted using an Internet Protocol Security Protocol (IPSEC) VPN, will go a long way toward preventing external attackers from even being able to access the application itself. Also, SSL and hypertext transfer protocol secure (HTTPS) encryption of Web transactions tunneled through that IPSEC VPN-encrypted network will help to thwart both resourceful attackers who manage to break through the VPN protection, as well as malicious insiders.

As with all IA protection mechanisms and DID layers the need for a given protection mechanism or layer must be driven by the assessed risk for the application in the specific environment in which it will operate. If the application is being developed for use in multiple different environments, it should be designed to be configurable to support the varying levels of security it may require in each operating environment or, if this is not possible, it should be designed to be secure in the highest risk of the intended target environments).

3.2 SECURITY-AWARE DEVELOPMENT

Security engineering of a software application should not be viewed independently from the rest of the application engineering life cycle. According to Andrew Jaquith of @stake in his February 2002 research report, “The Security of Applications: Not All Are Created Equal,” the overwhelming difference between secure applications and insecure applications is the use of superior development practices by the developers of secure applications when specifying, designing, coding, and deploying their applications. Good development practices, according to Jaquith, are more important than any security advantages bestowed by the choice of programming language or COTS software components such as Web servers, database servers, and middleware used in the application—all of which can be used in either a secure or insecure manner, but which are more likely to be used securely if the overall approach to development is security aware.

What follows are descriptions of security-aware development practices that will help strengthen rather than imperil application security.

3.2.1 Use a Security-Oriented Development Process and Methodology

The software development process and life cycle methodology you use can go a long way toward helping ensure that the applications you develop are secure.

The *Information Assurance Technical Framework (IATF) Release 3.0* (September 2000), published by the National Security Agency's Information Assurance Solutions Technical Directors, describes an Information Systems Security Engineering (ISSE) process that can be adapted to serve as a methodology for guiding a secure software engineering effort. IATF defines the ISSE as the process for addressing the user's information protection needs within the systems engineering, systems acquisition, risk management, certification and accreditation, and life-cycle support processes. The major ISSE systems engineering activities recognized by the IATF are as follows:

1. *Discover needs of the mission or business:* Determine the user's mission needs, relevant policies, regulations, and standards in the user environment
2. *Define system functionality:* Define what the system is going to do, how well the system must perform its functions, and what the external interfaces for the system are
3. *Design the system:* Construct the architecture; do specification of the design solution
4. *Implement the system:* Procure or produce, and integration of all components for the designed system
5. *Assess effectiveness:* Examine two major factors: (1) does the system meet the needs of the mission and (2) does the system operate in the desired manner of the mission organization?

The ISSE process is intended to be the subprocess of the overall systems engineering process that focuses on information protection needs, and which ideally occurs in parallel with the systems engineering processes. ISSE supports the evolution, verification, and validation of an integrated and life-cycle balanced set of system product and process solutions that satisfy customer information protection needs. The ISSE process also focuses on identifying, understanding, and containing information protection risks.

Another methodology that addresses the development process is International Standards Organization (ISO) 17799, "Information Security Management Certification." It includes a section on system development and maintenance that discusses, among other topics of interest (including security in application systems), security in the development and support processes for applications. The objectives of this section of ISO 17799 are to

1. Ensure that security is built into operational systems
2. Prevent loss, modification, or misuse of user data in application systems
3. Protect the confidentiality, authenticity, and integrity of information
4. Ensure that IT projects and support activities are conducted in a secure manner
5. Maintain the security of application system software and data.

Also of interest is the Systems Security Engineering Capability Maturity Model (SSE-CMM), discussed in Section 3.2.1.1 below.

Numerous commercial and government organizations have produced and published software assurance, software risk management, and secure software engineering and secure development methodologies (see Appendix B). By and large, these methodologies focus on the backend of the life cycle, that is, the certification and accreditation process, and ignore or only lightly touch on the front end of the lifecycle: the specification, design, and development of secure systems. The DoD Information Technology Security Certification and Accreditation Process (DITSCAP), for example, provides a thorough approach to the certification and accreditation of systems, but it provides nothing in the way of a methodology for developing those systems. The Common Criteria (CC), too, represents an extremely detailed, extensive set of criteria and processes for determining the security of the finished product, but it does not provide any kind of guidance on the best way of developing that finished product to ensure that it will satisfy the CC's evaluation criteria.

This said, these methodologies can be very useful both in helping define the security requirements—particularly the assurance requirements—for the security mechanisms and capabilities of the Web application, and in evaluating any development methodologies or adapting a methodology. The result would be to ensure that the methodology used is fully consistent with and supportive of the test, assessment, and documentation needs of the eventual accreditation and evaluation process that the application will have to undergo.

3.2.1.1 SSE-CMM

Recently, the Software Engineering Institute's Capability Maturity Model was used as the basis for defining the new SSE-CMM. The National Security Agency (NSA) began the effort to develop a CMM for security engineering in 1993 and eventually enlisted over 50 government, industry, and academic organizations to develop the SSE-CMM and its appraisal methodology.

Just as the SEI CMM is intended to help organizations control the processes they use for developing and maintaining software, the SSE-CMM provides guidance on how to gain control of the processes specifically involved in developing and maintaining trusted software. The SSE-CMM should enable organizations to improve their trusted software development processes and capabilities and to evolve toward a culture of security engineering excellence.

The SSE-CMM can also be used to assess developers of trusted software to estimate their level of ability, and to provide developers, vendors, and integrators with a clear picture of their current processes, their strengths and weaknesses, and needed areas of improvement. Use of the SSE-CMM should make it possible to simplify process for certifying and accrediting trusted application components by incorporating the necessary groundwork throughout the development life cycle that will encourage and ease development (and documentation) of creditable systems. In addition, its use should make it easier to qualify and rank bids during the acquisition of third-party trusted components to be used in applications.

For more information on SSE-CMM, see <http://www.sse-cmm.org/>. For information on other CMM variants, see <http://sepo.spawar.navy.mil/sepo/CMMVariants.html>.

3.2.2 Adopting an Effective Security Philosophy

“That which is not expressly permitted is forbidden.” This is the policy that you should follow when developing applications. If using a third-party software module, disable (and if possible remove) all capabilities of that module that you are not expressly intending to use. When writing your own code, write processes with only one entry point and one exit point, and exclude from the run-time environment all software libraries, routines, and others that you do not explicitly call from your application. Furthermore, assign only the absolute minimum of privileges to your application processes that they need to access the data or call the processes they need.

3.2.3 Planning the Development Effort Realistically

The application development project manager should write and implement a software development plan that includes both a contingency plan that addresses worst-case scenarios, and a development schedule that provides adequate time for disciplined software development, testing, and the inevitable delays caused by unanticipated problems.

3.2.4 Security Quality Assurance Throughout the Life Cycle

Thorough security quality assurance throughout the software life cycle will help guarantee the correctness and effectiveness of the security controls used by the application. If your development organization has no dedicated security quality assurance team, an ad hoc team should be assembled within your software quality assurance organization, to independently assess the application throughout its development life cycle. This team will identify, track, and provide developer guidance to eliminate the application’s security defects as early in the life cycle as possible.

3.2.5 Accurate, Complete Specifications

It can be difficult to determine whether an application security vulnerability stems from a faulty specification or faulty implementation. Software requirements and design specifications are notoriously vague and often result in poor implementation decisions. Specifications are also too often silent about how to address and avoid known security problems.

The first step in security-aware development is to write a detailed, accurate software requirements specification that identifies what the application needs to protect (such as data and configuration files), from whom, and for how long. The requirements stated in the application’s requirements specification should reflect, not conflict with or ignore, the organization’s application, data, network, and operational security policies. Part of the process of writing the requirements specification is identifying all policy statements that are relevant to the particular application being specified, and along with the interpretation of those policy statements into requirements statements that describe the specific ways in which the application must behave, in order to comply with policy.

After the requirements specification is approved, analyze the requirements it contains, and define in the application’s design specification the technical security architecture, controls, and functions, include

details such as how each control and function should operate, its interfaces, and the data it inputs and outputs that must be incorporated into the application to satisfy all the specified requirements.

An effective, accurate design specification will make it easier to pinpoint potential security vulnerabilities in the design early in the development life cycle, and it will enable you to make proactive design and implementation decisions that can at best prevent, and at least minimize and constrain the impact of, those vulnerabilities. This is a much less expensive approach than waiting to detect and address the application's vulnerabilities reactively during testing.

Update the software specification documents throughout the application's development life cycle, every time the application requirements, design, or implementation changes. This will help guarantee that the developers who maintain the application will have an accurate, up-to-date picture of the software they are responsible for keeping secure.

3.2.6 Secure Design

Software that does not perform correctly is not secure. But application security is achievable only when you truly understand the application you are building and carefully consider the following:

- Environment in which the application will run
- Application's input and output behavior
- Files and data it uses
- Arguments it should recognize
- Signals it should detect
- Nature of its interfaces to other applications and systems
- All other aspects of the application's behavior.

When designing the application, list all predictable errors that could occur during application execution, and define how the application will handle each error. In addition, address how the application will behave if confronted with an unanticipated error.

Before writing a single line of code, write a comprehensive code specification in language that is clear and direct. In short, write it in a way that would enable you, if you were to suffer amnesia, to use the document to continue your work on the application without difficulty. Specifically, make sure the application's design follows the guidelines that follow.

Consider the use of an application development middleware framework to minimize the likelihood of security problems being introduced through improper integration of application components. See Section 7.1 for information on application development middleware.

3.2.6.1 Minimize Functionality

If you follow the policy of "That which is not expressly permitted is forbidden," applications should contain only those functions they actually require to accomplish their purpose. Do not simply include

functions that will not be invoked, but which might be useful at some later date. In the interim, these unneeded, unused functions may be discovered by attackers and exploited. Also, because no one expected those functions to be used, they will probably not be audited, so the attackers can get away with exploiting them undetected.

When designing the application, explicitly specify the actions that any piece of code will be allowed to perform. Do not write processes to perform actions willy-nilly, but limit your code to do the following:

1. Perform only the actions you expressly define for it
2. Make only system calls to processes it absolutely needs to invoke
3. Execute only one task at a time
4. Initiate a new task only after the previous task has completed
5. Include only one entry point and one exit point (this is true for the application as a whole, and for each module and process within it)
6. Access only data it absolutely needs to successfully perform its tasks.

3.2.6.1.1 Minimizing Database-Related Functionality

Because Web clients (browsers) are untrustworthy, and because connections between browsers and servers are often easily compromised by attackers—and specifically because of the risk of SQL injection attacks—it will always be difficult and frequently impossible to establish and maintain a real trusted connection between the client through the Web server and extending to the backend database. Yet, this trusted connection is required to ensure the security of the backend database when it is updated by the client.

For this reason, Web front ends to databases should be read only (query only), that is, designed to submit and answer HTTP queries, but not to allow users to directly submit SQL updates (i.e., write to) to the backend database via the Web front end. If users must be allowed to update a backend database via the Web front end, the Web front end should be developed so that it validates all HTML forms submitted by the user. Thus, only valid data are accepted from the proper fields in HTML forms. An effective way to accomplish this validation is to translate the user's HTML form (which contains the data that will be used to update the database) into an extensible markup language (XML) update. Then use standard XML APIs and COTS parsers to validate the update data before passing them to the relational database management system (RDBMS) vendor-supplied resource access layer (between the Web server and the backend database) by which the XML update will be translated into SQL syntax and submitted to the database.

All Web-originated database updates must be executed as transactions to preserve data integrity in the database. For example, in a Java J2EE application server, use the Java Transaction API (JTA) with the

J2EE server to atomize updates extracted from user-submitted HTML forms into individual self-contained SQL database transactions.

NOTE: Java/Java 2 Enterprise Edition (J2EE), Enterprise JavaBeans, and Java servlets are the standard technologies used on the Navy's Task Force Web (TFWeb) Enterprise Portal.

3.2.6.2 Minimize Component Size and Complexity

Use multiple small, simple, single-function application components instead of one large, complex application component that performs multiple functions. Each single-function component should be atomic, so that it can be disabled when not needed or found to be vulnerable or errored without affecting the operation of other components. Moreover, atomicity and simplicity will make the components easier to understand and document, thus making it easier to verify their security and correctness.

3.2.6.3 Minimize Trusted Components

Trustworthiness implies that something deserves to be trusted. Unfortunately, application components and input are too often trusted without first making sure they are truly trustworthy. In many cases, there is simply no alternative except to trust certain components, particularly COTS components, without proof of their trustworthiness.

For this reason, applications should limit trust to those components that are absolutely critical to the application's secure operation. Specifically, the application should limit its trust to those very few components that perform the application's security control functions (e.g., I&A, access control and audit). All other components should be treated as untrusted by the application, and the application should be designed to minimize the potential impact of any security breaches caused by those components, and to ensure that the components have no access to security-critical information, functions, or privileges.

Similarly, data that come from application users should never be trusted by the application. The application should be designed to validate user input to prevent erroneous or malicious input from polluting the application or otherwise causing a security breach.

3.2.6.4 Minimize Interfaces and Outputs

Keep user interfaces as simple as possible. Provide only the functions needed, and make the interface nonbypassable, that is, make it impossible for the user to get around the interface to directly access data or protected functions. As noted in Section 3.1.4.4, applications should always minimize the amount of trust they place in the user and should never accept user input without first validating its correctness and benignity.

3.2.6.5 Avoid High-Risk Web Services, Protocols, and Components

Web services and application components that are the frequent subject of Computer Emergency Response Team (CERT) and other vulnerability reports, or which are simply widely known to be problematic, should not be included in DoD Web applications. If a high-risk Web service or protocol is absolutely required, it should be used only after having a security wrapper or execution “sandbox” applied to it to limit the potential damage it may incur if it misbehaves.

3.2.6.6 Disable or Remove Unused Capabilities and Resources

Unless a function (within the application) or resource (e.g., libraries or data files external to the application) is invoked or used during application processing, if at all possible, disable it or remove it completely from the application code and run-time environment. Unneeded functions and resources represent potential targets of attack. Narrow the number of available targets, and you reduce the number of potential attacks.

3.2.6.7 Separate Data and Control

Keep files created by the application should completely separate from programs executed by (or within) the application.

If the application absolutely must accept programs that are remotely downloaded, implement a very restrictive sandbox in which the downloaded program is allowed to run. Make sure the sandbox prevents the program from bleeding into other areas of the application’s execution environment.

If you must include auto-executing macros (i.e., macros that execute when the application is loaded or when the data are displayed, or both) in files created by the application, fully test those macros to ensure that they do not create security vulnerabilities, and execute the macros in a “sandbox”.

Be aware that sandboxes are far from being a complete solution. They are easily exploited by hackers and malicious programs. For this reason, in addition to sandboxing, store all programs in separate files so they can be blocked more easily if a sandbox vulnerability is discovered. Such separate program storage has the added benefit of making it easier to cache and reuse the program.

3.2.6.8 Protect All Sensitive Transactions

All Web applications should use SSL/TLS (i.e., transport layering security) (SSL Version 3.0 or TLS Version 1.0) with approved cryptographic and key management algorithms to implement seamless end-to-end session encryption of all their network-based transactions in which sensitive information is transmitted. In addition, Web applications should implement hash or digital signature to ensure the integrity of transmitted data. When necessary, the application should also implement digital signature to ensure accountability (through nonrepudiation) of transmission, manipulation, and receipt of transmitted data.

3.2.6.9 Protect Sensitive Data at Rest

The application should ensure the safe, secure handling of sensitive data at rest as well as in transit. This protection includes confidentiality, integrity, and availability protections. The mechanisms for protecting data at rest include the underlying access controls of the Web server or backend database in which the data are stored. In situations where those access controls are considered inadequate to protect the data, encryption of the data before storage may be implemented to augment the access controls.

3.2.6.10 Include Trustworthy Authentication and Authorization

Include reliable, trustworthy user authentication and authorization. Do not use Web server basic authentication without also using HTTPS and SSL/TLS to create an encrypted pipe for transmitting passwords or session IDs over the network. Also make sure the application ensures that the passwords, encryption material, and whatever else it uses are adequately protected when at rest (i.e., stored on the Web server).

3.2.6.11 Always Assume the Operating Environment Is Hostile

Always assume that the application will run in the most hostile environment possible, and code it to protect itself. That way, if the application is ever ported to another system or moved to operate in another environment, or if some part of the infrastructure (or the application's own middleware or platform) undergoes an upgrade that introduces vulnerabilities not found in the current version, the application will not suddenly become vulnerable due to now-obsolete assumptions about a safe or protected environment.

Security mechanisms in the operating environment should be seen only as an additional layer of protection around the application (i.e., DID); they should never be considered as replacing the need for the application to protect itself.

In addition to tending to the network and platform layers of security, implement an application layer of DID, so that the application never becomes vulnerable due to reliance on a single protection mechanism in its surrounding infrastructure or underlying operating platform. For example, write a PK-enabled application so that, were underlying PKI to fail, the application would shut down (in an orderly, secure manner) rather than continue to operate without cryptographic services. Thus, the failure of the PKI, as the result of a denial of service attack, for example, could not compromise the application itself or the data it processes.

3.2.6.12 Always Assume that Third-Party Software Is Hostile

Beware of third-party (both COTS and open source and other public domain) software and common software libraries. Never trust anyone else's software without adequate proof of its reliability and security. Thoroughly review the specifications of third-party products before using them, and thoroughly test any third-party components that will perform any trusted (privileged) functions within or on behalf of your application, such as encryption, authentication, and sensitive data access. Follow the guidelines in

Appendix C when selecting third-party software, particularly software that implements security-sensitive functions.

Write your application to strictly validate all inputs it receives not just from users, but from any external processes (i.e., processes in third-party components), to ensure that the input data cannot induce a buffer overflow in the application. Even if your application code is written in a language that is not vulnerable to buffer overflows (e.g., Java), be very careful about any C or C++ libraries called by your application, because the library routines may be susceptible to buffer overflows even if your own code modules are not. Research the security track record of the third-party modules, as well as the C/C++ library routines you plan to use. For example, check for CERT and other reports of vulnerabilities and hacker attacks involving these modules or routines.

3.2.6.13 Never Trust Users and Browsers

All Web application designs should be predicated on the understanding that neither user input nor browser operation is trustworthy. All sensitive functions, including extensive user input validation, should be performed by a trustworthy server application, and not in the browser.

3.2.6.14 Require and Authorize No Privileged Users or User Processes

Do not write the Web application to assume or require that it will be granted anything more than absolute minimum (user) privileges. If a trusted process within the application must be granted elevated privileges, do not allow this process to be invoked by a user or user-controlled process. Isolate all privileged processes from any potential control or compromise by users. The exception may be the installation and configuration routines that enable the administrator to configure the application's initial operating state, any functions involved with configuring, reading, and archiving the application's event logs. Write the application so that its administrator functions can be performed only by a strongly authenticated administrator and cannot be accessed (or even seen) by other users. See Section 4.3.3 for a discussion of least privilege in applications.

3.2.6.15 Do Not Rely on Security Through Obscurity

Do not rely on security through obscurity. Hiding information such as security data and configuration details may seem like an effective subterfuge, but it is not the same as adequately protecting that information. Security through obscurity is predicated on the hope that no one will accidentally or intentionally discover the sensitive information—a faulty assumption at best, and disastrous at worst.

Obfuscation may be effective when and only when used in conjunction with proper security protections, to help discourage certain types of attacks by unsophisticated nuisance hackers and so-called script kiddies (versus sophisticated attackers and e-warriors). But as a true security measure, obfuscation is unreliable, ineffective, and frankly should be unnecessary if the required security protections are implemented correctly.

A specific area where obfuscation is often promoted is in the instance of Java programs, which due to their simplicity and portability have proven susceptible to reverse engineering. Java bytecode

obfuscation is often promoted as a protection against this type of reengineering attack. However, in the DoD environment, the DoD mobile code policy requires that all Java code be digitally signed before release, and that the code signature be validated by the user before the code can execute on his client system. This code signature and validation not only enable the user to verify that the code comes from a trustworthy source and has not been tampered with, but they pretty much eliminate the possibility that the Java code could have been captured and reverse engineered before being served to the unsuspecting user. At least, the code signature and validation provide the user with an opportunity to explicitly decide whether to run any Java code on which a signature has not been applied or validated, or both. So although Java bytecode obfuscation may seem like a good protection measure, it is superfluous in a DoD Web application environment.

3.2.6.16 Be Accurate in Your Assumptions About the Underlying Platform

When the application has to rely on or use security features and services provided by the underlying platform or operating environment, make sure that the assumptions you make about the operation of those security features, and the application's interfaces to those features, are correct. If the application may be ported to unanticipated platforms, be sure that the application has been written, and is configurable, to handle differences in the security environment and services provided by the underlying platform.

Whenever possible, take advantage of tools that enable the application to leverage the existing platform and network security infrastructure.

3.2.6.17 Make Security Mechanisms Easy to Configure and Use

Make configuring the application's security features as easy and clear as possible on both the server and the client, including post installation configuration. Make secure use of the application as easy as possible to prevent administrators from taking short cuts or prevent users from attempting to shut off or bypass security controls.

Design all user interfaces to application security mechanisms (e.g., digital signature and encryption tools) to be easy to use, so that users are less likely to try to bypass those mechanisms and more likely to use them correctly. Try to design the user interface to each mechanism to match the user's mental image of his or her goals in using the mechanism, that is, make their use intuitive. The same is true of the administrator's interface: a well-designed interface will reduce the likelihood of the administrator incorrectly configuring the application's security mechanisms.

3.2.6.18 Risk Analysis of Application Design

Before submitting the application's design for review, perform a thorough risk analysis of the design. The risk analysis should identify all potential threats to the application, and rank them according to severity and potential impact. It should identify any residual vulnerabilities in the application's security posture and identify the changes to the design required correct them. Finally, it should include an

estimate of the cost of implementing each identified change. Note that a design defined by a security-aware developer will be less likely to contain vulnerabilities to be revealed by the risk analysis.

3.2.7 Application Middleware Frameworks

The objective of application middleware is to create an application-layer framework that implements a networking and security infrastructure that can remain consistent from one application to the next. The result can also minimize the amount of custom-development and integration required to provide security services, communications services, and other standard services that all applications need. Application middleware frameworks are discussed at length in Section 7.1.

3.2.8 Restricting the Development Environment

As the development life cycle progresses, and the code moves into a preproduction environment where accounts and permissions are managed properly, numerous problems may suddenly appear in the software's operation. Those problems were not evident in the unrestricted early development environment because various functions were allowed to operate without any constraints in the development environment. For example, data that were read and write accessible in the development environment may no longer be accessible in the operational 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.

In a distributed application, just identifying the root cause of problems originating from the sudden application of restrictions will be a challenge, and that will add to the time it takes to debug and test the application. For this reason, it is better not to develop code under the administrator or any other privileged account on the development platform. Developing code unconstrained by access restrictions allows the code to do anything, anytime. Although this makes it easier to write and test functionality, it also prevents you from seeing the security impacts and implications of your design choices and the functionality you have implemented. Use of the administrator account also prevents application isolation, accurate testing, and accountability. As the ostensible administrator you have access not confined to a specific application, so you inadvertently overwrite other developers' code and data, particularly if those developers give their files (or, in the case of databases, table names) names similar to those you have given to your files and tables.

Security-aware development means creating and using application-specific accounts with rights and privileges that reflect those that will be assigned in the operational environment; in this way, the development environment will reflect reality to the greatest extent possible. Of course, you may have to refine these access rights over time, but if they are not defined in the first place, it will be impossible to know how to refine them when it is time for the application to be tested.

3.2.9 Writing Elegant Software

Elegant software is more correct, and more secure than other types are. What follows are some tips on writing code in a way that will yield elegant software.

3.2.9.1 Document First

As noted in Section 3.2.6, you should write the requirements, design, and code specifications before writing a single line of code. The proactive documentation of applications is not only a good development practice, it is imperative for systems that must undergo certification and accreditation under the DITSCAP or Director of Central Intelligence Directive (DCID) 6/3. Furthermore, after the application is documented, it should be implemented and coded to conform strictly to its design and code specifications. Applications that do not match their specifications are incorrect and insecure by definition.

3.2.9.2 Keep Code Simple, Small, and Easy to Follow

Such direction is particularly true for code that implements trusted, critical, or otherwise sensitive functions. Use structured programming, and avoid recursions and *goto* statements that blur the flow of control.

NOTE: Structured programming is a technique for organizing and coding programs in which a hierarchy of modules is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used in structured programs: sequential, test, and iteration. The fundamental principle of structured programming is this: At all times and under all circumstances, the programmer must keep the program within his intellectual grasp. Well-known methods for achieving structured programming are: (1) top-down design and construction, (2) limited control structures, and (3) limited scope of data structures. Structuring data for intelligibility means limiting the scope of variables, explicitly declaring all variables, using meaningful data names, and using hierarchical data structures.

You will greatly reduce the likelihood of bugs in your code by reducing the amount of code you write. Elegant software is efficient software: it implements each function by using the minimum number lines of code possible. In using elegant software, also remove all unnecessary software from the code base. Avoid ambiguities and hidden assumptions. Also remember: the smaller and simpler the code, the easier the application will be to accredit.

3.2.9.3 Isolate Security Functionality

Place security-critical functionality in separate modules that are simple, are precise, and have been proven to operate correctly. Write all modules—even small ad hoc scripts—so that they do not rely on any global state; this measure will avoid unnecessary complexity and ensure that the module's flow of control can be tracked.

Use only precisely defined interfaces for communication between those modules. These interfaces should not directly reference internal variables.

As stated earlier, the application's design should constrain trust to only a very few components in the application, rather than distributing it widely across numerous components. However, this minimization of trust should not be taken to its extreme. Trust should not be completely centralized into a single component—an extreme that is contrary to the idea of DID—because doing so creates a single point of failure and a single high-value target for attack.

3.2.9.4 Be Careful with Multitasking and Multithreading

Unless the application program runs on a multiprocessor machine, write programs to be single-tasking (i.e., to do only one thing at a time) unless there is a good reason for them to multitask. Multitasking and multithreading programs in operating systems that support multitasking and multithreading can improve application performance, but they also increase application complexity. Complexity is the enemy of security because it makes the application harder to understand, to analyze, and to verify in terms of security and correctness. Multitasking also increases the risk of deadlocks, which occur when two tasks or threads both stop executing and wait for one other to terminate.

If your program will do multitasking or multithreading, carefully analyze its operation to be sure that the simultaneous processing of tasks and threads does not create conflicts in usage of system resources (such as memory or disk addresses). Synchronize the tasks and threads to prevent such conflicts. As with all structured programs, write each task to contain only one entry point and one exit point.

3.2.9.5 Use Secure Data Types

Be careful about what data types your program uses, especially in its interfaces. For example, signed and unsigned values are treated differently in many languages (such as C or C++). Consider the security implications of each data type you use, and avoid data types that are likely to increase the vulnerability of data to compromise.

3.2.9.6 Reuse Proven Secure Code

Whenever possible, reuse previously debugged, tested, security-validated application components and software libraries, instead of writing new software. Use and reuse accredited or evaluated trusted components whenever possible. If a component has not been accredited or evaluated, it should undergo a thorough risk analysis, including some level of security testing, before being accepted for use in the application.

In all cases, thoroughly analyze the code—whether it is GOTS, COTS, or open source, and whether accredited or not—to determine whether the code includes assumptions about its runtime environment that are not true in your own application's operating environment (such as whether it makes system calls to nonexistent routines or programs or includes environmental variable settings that are in conflict with your run-time environment). If any such assumptions exist, determine whether the code can be cost-effectively rewritten to change or eliminate these assumptions, or alternately, whether a security wrapper can be applied to isolate the reused module so that its assumptions will not affect the overall

application's operation. If such changes cannot be made cost-effectively, the code is not a good candidate for reuse.

3.2.9.7 Use Secure Programming Languages and Development Tools

Choose a programming language that supports good coding practices, that does not have lots of inherent vulnerabilities, and that can be used securely. As popular as they are, C and C++ are probably the least secure languages that you could use, particularly when compared with Java, Python, and other languages that are inherently resistant to buffer overflow. Investigate these alternatives and, if at all possible, use them.

If for performance or integration reasons, you must use C or C++ for some or all of the application, be scrupulous about coding the necessary validations to avoid buffer overflows; the language itself will not automatically protect against buffer overflow. Whatever language you use, avoid all calls and commands that are known to have security problems (see Section 3.2.9.8.1), and avoid the language's more obscure, unfamiliar features.

Also, use development tools that force you to follow good software engineering practices, for example, that force you to create documentation before writing code and that highlight bugs in your code so that the code can be rewritten to eliminate them.

3.2.9.8 Call Safely to External Resources

Self-contained application programs are rare. Most programs call out to system-level programs or middleware. To ensure that calls from application programs to other programs are done securely, applications should be implemented with the following limitations.

Nearly every programming and scripting language allows the use of system-commands to pass input (commands) to the underlying operating system. The operating system executes the given input and returns its output to the application along with various return-codes indicating whether the requested command was executed successfully or not. In Web applications, system commands are often used for file handling (delete, move, and copy), to invoke system programs such as *sendmail* or file transfer protocol (FTP), or to call operating system tools to modify application input and output (i.e., filters).

3.2.9.8.1 Use Only Safe System Calls

To avoid availability and other security problems, before using any system call take the following steps:

1. Check the arguments passed to operating system functions in the call to be sure they contain the expected parameter values.
2. Check all return codes from system calls (in C and Perl). If a call fails, check the *errno* variable to determine why it failed; if the reason is unexpected, write the application to log the unexpected value, delete all temporary files, and then gracefully terminate itself. This approach

should help in tracking down any programming bugs or security problems that might be causing the failure.

3. Write UNIX applications to call `fork()` to launch new processes. Do not call the less secure `vfork()`.

Refer to Appendix D for system calls to be avoided in individual programming languages. Many of the following recommendations also form the basis for the input validations your application should perform on user-supplied data. See Section 5.3 for information on how to implement input validation.

3.2.9.8.2 Call Only Trustworthy Library Routines and Interfaces

Before using any library routine or programmatic interface, read its specifications and test it to ensure that it performs as specified and does not violate system or application security. If any routine or interface is found to be insecure, that is, untrustworthy, write or obtain a new, trustworthy version of the routine or interface to replace the untrustworthy version. If you doubt that future versions of a currently trustworthy routine or interface might not be trustworthy, or if a routine or interface that is safe on one platform cannot be guaranteed to be equally safe on the other platforms on which it must run, replace the questionable routine or interface with a new routine or interface that is implemented securely.

3.2.9.8.3 Accept only Valid Values in Parameters

The application call to another program should permit only valid, expected values for all parameters. Many library calls and commands call lower-level routines indirectly by calling the shell. However, passing in parameter characters that happen to be shell metacharacters can bring unexpected and undesirable consequences.

3.2.9.8.4 Exclude Metacharacters

Many programs, including command line shells and SQL interpreters, include metacharacters in their input. These metacharacters are interpreted as actionable instructions, rather than as passive data: metacharacters may indicate commands, or delimiters intended to separate commands or data. If the application program invokes an external program that handles metacharacters, the application code may be subject to hacker insertion of metacharacters as a hijacking mechanism. Refer to Section 5.3.4.3.2.

3.2.9.8.5 Include Full Pathnames Only

Application calls to external programs, as well as other application references to files, should always specify the full pathname of the external program and file (e.g., `/usr/bin/sort`), thus eliminating the possibility of the application calling the wrong program or starting from the wrong directory.

3.2.9.8.6 Use Only Interfaces Intended for Programs

Applications should call only external programs and APIs intended for use by programs; applications should not call interactive (user) programs or APIs to such programs.

3.2.9.8.7 Validate All Data Returned from System Calls

System calls that can return error conditions should be validated by the application to ensure that the returned error conditions cannot be exploited to cause resource exhaustion (e.g., via multiple simultaneous calls from CGI scripts or other server programs). If the system itself cannot handle a particular error condition gracefully, the application must be written to fail safely when detecting that error.

3.2.9.9 Use Escape Codes with Extreme Caution

3.2.9.9.1 Command-Line Escape Codes

Application programs, especially those with user interfaces, often include escape codes that perform functions such as invoking the command line. For example, some command line mail programs, such as *mail* and *mailx*, use the ~ (tilde) as an escape character that can invoke a number of commands. Interactive programs, such as *vi*, *emacs*, and *ed* on UNIX systems, often include escape codes that allow users to run arbitrary shell commands from within their sessions.

Escape codes should be included in the application only if the program being escaped to is trustworthy. The problems inherent in escape codes in interactive programs can be avoided by having the application call only to programs that are explicitly intended for use by other programs.

3.2.9.9.2 Device Escape Codes

A particular set of escape codes, those in the Hayes modem command set, may be inserted into programs to launch denial of service attacks against modems or to connect a user to a different modem. If the application issues a call that controls a modem (or another lower-layer device or emulator), all escape codes for that device should be excluded from the application. Similarly, if the application invokes a terminal interface, it should exclude any escape codes for terminal types, such as VT100, because such escape codes may allow denial of service or other attacks that involve sending untrusted data directly to a terminal screen. If data are displayed to users via (simulated) terminals, those data should exclude any control characters (i.e., characters with values less than 32) if the user receiving the data cannot be proven trustworthy.

Emulated terminals, called *tty* or *ttyp* in UNIX, an abbreviation of the word *teletype*, should be writable by their owners. The other write permission should not be set for emulated terminals, nor should the group write permission. In addition, the user group for the *ttyp* should contain only the owner implementing the user-private group scheme.

3.2.9.10 Maintain a Consistent Coding Style

Maintain a consistent coding style throughout the application, regardless of how many developers actually write the code. This includes the appearance. For example, have all developers use the same, consistent approach to indenting lines of code. Individuality of style is not a desirable quality in application code.

3.2.9.11 Find and Remove Bugs

Many security problems are caused by programming errors. But also recognize that, from a security standpoint, there is little difference between a software failure caused by a run-time error (bug) and a software failure caused by a user error—except that the latter is likely to be the result of a poorly designed or errored user interface. Be careful about how user interfaces are designed and implemented. As noted earlier, do not make them counterintuitive or overly complicated.

Debug application code carefully and thoroughly before testing, and develop test plans and scripts that will exercise the application thoroughly, increasing the likelihood that any residual bugs will be discovered. Testing of security-sensitive functions and modules should involve penetration testing as well as correctness testing (see Section 3.10). Removing bugs to make programs more secure has the added benefit of making them more reliable as well.

3.2.9.12 Write for Reuse

Comment on, document, and test all code you do write with reuse in mind. That is, make the code's purpose and functionality obvious to other developers who may consider reusing it.

3.2.9.13 Keep Third-Party Component Fixes and Security Patches Up to Date

No matter how secure your development practices are, you cannot prevent vendor-induced implementation vulnerabilities in COTS software for which you do not have the source code. Therefore, one of your secure development practices must be to keep all third-party products up to date in terms of applying bug fixes and security patches.

That said, general functional updates to third-party software may actually introduce new security vulnerabilities. Do not update software simply for the sake of being current if the new functionality is not needed by or will not be used by your application.

If the vendor plans to discontinue maintenance of an older product version including no longer providing fixes and patches in response to customer-identified problems, you need to discuss this with the procurement officer or contracting officer who established the original contract with the vendor. To determine if anything can be done to influence the vendor to continue supporting the software for you, or to release the source code for the old version to you. Otherwise, if you must use the updated software to continue getting patches and fixes, obtain a copy of that software and analyze and test its new features thoroughly to discover any security problems they may contain or introduce to your application, and encourage the vendor to fix these problems before you adopt the new software. If the software is found to contain security vulnerabilities that are too great to be tolerated, you may need to find functionally comparable but more secure software from another vendor.

3.2.9.13.1 IAVA and Security Patches

In compliance with the Deputy Secretary of Defense's 30 December 1999 Information Assurance Vulnerability Alert (IAVA) policy memorandum, DISA established an IAVA database of alert

notifications pertaining to vulnerabilities discovered in DoD systems and an IAVA Implementation Process to provide a means for the CinCs, Services, and Agencies (C/S/A) to report discovered vulnerabilities, to receive the latest vulnerability information reported by others, and to identify the latest available security patches for common operating environment (COE)-compliant products and systems.

Although the primary intended audience for the IAVA information is system administrators, as a developer, you should take advantage of IAVA process as you design and implement your Web applications, to ensure that the versions of third-party products used in your application are absolutely up to date in terms of available security patches. Please refer to <http://diicoe.disa.mil/coe/coeeng/iava.html> for further information on the IAVA Implementation Process.

3.2.9.14 Common Logic Errors to Avoid

Many logic errors arise from design flaws. Some are the result of poor implementation. Useful techniques for detecting common logic errors include

- Defensive programming language design
- Compiler checks
- Formal methods for analyzing the application program's conformance to its specifications
- Static checking, using a formal method, an informal method, or a code-scanning tool such as lint or LCLint (Appendix C lists several such tools). Strong type checking and formal static checking expose both consequential (securitywise) and inconsequential errors. It is up to the developer to distinguish between the two and to treat the security-relevant errors appropriately.

3.2.9.14.1 Inconsistent Naming

A common source of vulnerabilities in application code is the use of aliases, pointers, links, caches, and dynamic changes without relinking. To reduce the likelihood of problems

- Treat aliases symmetrically
- Use symbolic naming and dynamic linking
- Use globally unique names
- Clear caches frequently.

3.2.9.14.2 Improper Encapsulation

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

- Effective system architecture
- Effective programming language design

- Effective software engineering
- Static checking
- Dynamic checking.

3.2.9.14.3 Asynchronous Consistency

Timing and sequencing errors, such as order dependencies, race conditions, synchronization, and deadlocks, can threaten application operation. Many timing and sequencing errors are caused by sharing of state information (particularly real-time or sequence order) across otherwise disjoint abstractions. Applications should use the following techniques to avoid such errors:

- Atomic transactions
- Multiphase commits
- Hierarchical locking strategies.

3.2.9.14.4 Other Common Errors

Other common logic errors to be avoided include these:

- Off-by-one counting
- Omitted negations
- Absolute values.

3.2.10 Security-Aware Testing

Application functional security testing and penetration testing should be performed on the same platform that has been locked down according to FSO STIG, COE, GCCS, NSA, and any other relevant security guidelines, and generally configured as it will be when deployed operationally. To the greatest extent possible, the test environment should also duplicate the intended operational environment, in terms of the surrounding security infrastructure. Many systems have been tested on out-of-the-box platform configurations only to later fail to operate after deployment when security parameters and ownership/access protections on certain files were set to their proper operating values. The result has been the need for expensive security retrofits, or the insecure configuration of the platform so that the application can run – neither situation being acceptable.

3.2.10.1 Rules of Thumb for Security-Aware Testing

The objectives of security-aware testing are to verify that the software performs without errors, unintended interruptions, or failures, and that the application cannot be easily compromised. Here are some general guidelines to follow when testing your application:

1. *Think like a hacker.* Devise a test plan composed of hacker-like strategies and scenarios with which to attack the application. Include tests that attempt to cause buffer overflows, race

conditions, flood conditions, and other problems, as well as tests that involve entering obscure command line options.

2. *Submit unexpected input to the program.* Note how the program reacts, particularly whether it manifests any vulnerabilities or anomalies that could be exploited by a hacker.
3. *Stress test the application with junk input.* Randomly generate and send long strings of data in an attempt to overflow every single input field that gets parsed by a certain protocol. Observe whether overflow actually occurs, or whether the application handles the junk input correctly.
4. *Delay the program between two system calls.* Again note what happens, and whether an exploitable vulnerability or anomaly is revealed.
5. *Include independent verification and validation (IV&V) in your test plan.* Subject the application to independent testing by outsiders who have no preconceived expectations of how users will interact with the application, or how the application itself will perform.
6. *Verify compliance with the design.* Just as you ensured that the design satisfied the stated requirements, ensure that the software implementation conforms with the design.

For an overview of the issues involved with software testing, refer to <http://www.computer.org/software/so2000/pdf/s1070.pdf>.

3.2.10.2 Code Reviews

Periodically review your source code in progress throughout the development life cycle to identify and correct any obvious problems as early as possible and definitely before submitting the code for further testing. Thorough code review includes these measures

1. *Reviewing the source code* for the potential vulnerabilities identified in the software risk analysis.
2. *Manually reverse engineering the trusted modules of the program* by carefully reading the disassembly of the module code, to reconstruct the program flow and spot programming errors and potential vulnerabilities. *Investigating suspicious code constructs* while reverse engineering the code. Look out for calls to functions that are known to be the source of common programming errors and vulnerabilities. Review these calls and flag any that seem to cause security problems. For example, flag the use of any call in the C/C++ `scanf ()` family of calls if it is used to send data to a string without first specifying the maximum length of that string. *Using a code scanner* to identify common security error conditions, such as buffer overflows in C and C++.
4. *Questioning all assumptions* and choices made in the code, such as choices related to trust of data received from external users or processes.

5. *Checking the correctness* and validity of all command line arguments, system call parameters, and return values.
6. *Writing the code to be testable*. The following article from the March/April 2002 issue of *IEEE Software* magazine describes what the authors call “test-first design” and which includes several good recommendations for writing testable Web application code:
http://www.edwardh.com/ieee_article/goingfaster.pdf.
7. *Subjecting the code to peer reviews* throughout the development cycle. Allowing eyes other than your own to peruse your code may reveal problems or ambiguities early on that you may have overlooked and spare you from having to rewrite the code more extensively after testing. After the peer review, walk through the code with the developers who reviewed it, and explain to them what each component and each part does. Explaining why something is done in a certain way can surely help you to discover logic errors and lead you to reconsider why you implemented the code in a certain way.

3.2.10.3 Source Code Security Audits

Source code security audits are best performed by the security experts in your development team, the organization’s risk management or accreditation team, or an expert independent consultant. These audits should include both white hat and black hat audits.

3.2.10.3.1 White Hat Audit

White hat auditing reviews every line of code in the application, identifies fixes for every discovered problem, and as a result, leaves the program in a more secure, stable state. The audit continues after each vulnerability is found and documented. It should be repeated whenever the software is updated.

3.2.10.3.2 Black Hat Audit

Black hat auditing is performed with a hacker mentality. It looks at the code specifically to find the kind of vulnerabilities through which hackers often attempt to compromise application security. Only the suspicious parts of the code are reviewed, and the audit is repeated after all discovered vulnerabilities are fixed.

3.2.10.4 Penetration Testing During Development

Penetration testing of an application in development can do IV&V of security trustworthiness of the application and the strength of its incorporated and underlying security protections and mechanisms. Penetration testing can help find security flaws and complements security functional testing, which confirms the correct behavior of the application’s specified security features, functions, and capabilities (i.e., “what is there”) by testing “what is not supposed to be there.” There are no specifications for flaws in the application’s security implementation or other residual vulnerabilities, so such flaws must be discovered almost haphazardly. Still, a rigorous penetration testing methodology is likely to reveal more rather than fewer critical security flaws.

Penetration testing should be a part of the overall security plan prepared in the early phase of the development program, for example, by the system requirements review (SRR). Penetration testing should not be part of acceptance tests, which are designed to demonstrate that the application meets its specifications. Whereas, security functional testing can be included in acceptance tests, penetration testing is more useful as a design and implementation validation tool while the system is still under development. IT would begin as soon there is solid design evidence such as the architectural analysis of the application presented at preliminary design review (PDR), the flaw generation report presented at critical design review (CDR), and code test results from the system integration testing. The main objectives of penetration testing during development are (1) to determine the likely level of knowledge and experience (sophistication) an attacker would have to have to successfully compromise the application and (2) to identify additional security protections and countermeasures that could be implemented to both significantly elevate the attacker's necessary level sophistication and reduce the potential pool of attackers with the qualifications to succeed in compromising the application. In addition, formal penetration testing may be required after the formal security test and evaluation (ST&E), as part of the system's certification and accreditation (C&A) requirements.

The goals of penetration testing should be clearly stated in test plans before testing begins. A comprehensive penetration test plan identifies the object being tested and defines the sophistication and characteristics of anticipated attacks to be simulated by the test. The simulated attacks should reflect the results of the risk assessment of the application. That is, is it more likely to be targeted by a malicious insider, an external amateur hacker, a hostile country's infowarrior, or a cyberterrorist? The worst-case assumption is always a hostile attack from authorized malicious users who exceed their authorizations. Finally, the penetration test plan should define the limits and duration of the penetration test, that is, when the test will be considered complete. The open-ended nature of flaw searching may conclude when resources or testers are exhausted, or a time limit may be applied.

4.0 IMPLEMENTING SECURITY MECHANISMS

4.1 PUBLIC KEY-ENABLING

PK-enabling for Web applications means implementing the PKI that will support use of the HTTPS and SSL protocols that are the core of Web authentication and of confidentiality and integrity of data in transit.

SSL was designed to perform one-way authentication of Web servers to browsers, so that the user would know that he was connected to the expected, trusted server. SSL Version 3 also servers authentication of browsers (and, by implication, their users), through the SSL mutual authentication option that uses personal identity certificates for the user's authentication to the server. See 4.1.4 for more information on PK-enabling of Web applications.

We recognize that SSL does add a performance overhead to Web transactions that in some applications may seem significant. Depending on the Web protocol being used, the power of the server hardware, and the network environment, SSL connections have been observed to range from 2 to 100 times slower than unsecured transfer control protocol (TCP) connections. It is critical, therefore, that Web application designers be well aware of the transaction performance requirements for their applications and understand the relative impact on performance that SSL will add to each transaction; developers should choose the protocols they use accordingly, to minimize the performance impact of SSL. Also, it is likely that not every transaction will need to be SSL secured. If a transaction does not transmit sensitive data, it probably does not need to be SSL encrypted. In some applications, the need for SSL may be limited to the authentication dialogue. In others, SSL may also be needed to encrypt downloading of Web pages that contain sensitive data, or to upload of user input forms. In environments where the main threat is perceived to be external, and where infrastructure countermeasures such as VPNs, link encryption, and firewalls go some way toward preventing eavesdropping by outsiders, it may be adequate to transmit nonsensitive data without SSL encryption and limit SSL strictly to sensitive transactions and data. However, if there is any concern about an insider threat, the need to protect the application must be weighed carefully against the need for optimal performance. Finally, use of hardware-based SSL accelerators should be considered for applications and environments in which SSL will be used to secure a large number of transactions or in which significantly performance-affected protocols must be used. Appendix C lists some SSL hardware accelerators.

PK-enabling of non-Web applications, including legacy (and nonlegacy) backend systems interfaced with by Web server applications, will enable them to perform authentication, encryption, and digital signature, using the same PKI as for the Web application. PK-enabling of non-Web applications will likely entail use of a PKI toolkit. More information on PK-enabling of legacy applications appears in Section 4.1.5.

NOTE: The encryption technology specified for DoD Web applications is the DoD PKI. If an alternative encryption technology is desired, such as FORTEZZA in a particular operating environment or for a particular application component, then

a waiver must be obtained from and the choice of technology must be approved by the DoD application's owner and its accreditor; in all cases, the encryption technology used in DoD applications must be approved by NSA.

4.1.1 PK-Enabling: A Definition

A quick set of definitions of PK-enabling may be helpful in understanding when and how public key enabled application security services should be implemented. *Department of Defense (DoD) Medium Assurance Public Key Infrastructure (PKI) Public Key-Enabling of Applications* (29 September 2000) lists the following specific functions that PK-Enabled applications must perform:

1. Accept and process a DoD PKI X.509 digital certificate to support one or more application-specific functions (digital signature, data encryption support, system or network access) that provides security services
2. Include an interface to a hardware token supported by the DoD PKI
3. Collect, store, and maintain the data required to support the use of signed data in a security service
4. Interoperate with the DoD PKI as verified by the Joint Interoperability Test Command (JITC) Department of Defense Public Key Infrastructure Interoperability Generic Test Plan (Version 1.1, May 2001).

The specific security services for which an application may be PK-enabled include the following:

- *Authentication:* With PK-enabling, the PKI replaces (or in some cases augments) the existing authentication system based on username and password or IP address filtering, with an authentication system that uses digital identity certificates as the authentication factor.
- *Confidentiality:* PK-enabling allows the application to use PKI technology, generally in conjunction with standard symmetric encryption, to encrypt data in transit or at rest, or both. In virtually all implementations of PK-enabled data encryption, the PKI is used to securely distribute the symmetric secret keys over unprotected networks; it is the symmetric keys that will be used to encrypt the data. The security of the key distribution is achieved by using an asymmetric (PKI) cryptkey pair (public and private keys) to encrypt the symmetric cryptkeys before distribution. In many implementations, including the DoD PKI, the cryptkeys (asymmetric and symmetric) are stored in a secure repository (directory), and retrieved when they are needed via the PKI's lightweight directory access protocol (LDAP) capability or, in some cases, X.500 DAP.
- *Integrity:* Implements PKI technology to calculate and affix cryptographic hashes to executable code and data. These hashes are then used prove that the code and data have not been tampered with between the time they were created and the time they are executed or accessed.

The PKI may also be used to digitally sign code and data to prove that they originated from a known, trustworthy (or at least trusted) source.

- *Nonrepudiation:* Implements PKI technology to digitally sign, in an irrefutable, legally enforceable manner, transactions and documents.
- *Authorization:* PKI alone cannot provide authorization. However, X.509 Version 3 digital certificates accommodate inclusion of attributes that may indicate user roles, rights, or privileges. When used in conjunction with a directory service or access control list, these X.509 attribute certificates can constitute a PKI-based authorization service.

The *Public Key Enablement Protection Profile* (Draft Version 0.2, March 22, 2002), developed by Entrust CygnaCom for the U.S. Marine Corps, provides a useful set of security metrics for application developers, to ensure that their PK-enabled application provide at least the core set of functions required to establish medium assurance.

4.1.2 Why PK-Enable?

In environments with VPNs it may at first seem redundant to PK-enable the application. VPNs, including those based on IPsec, provide cryptographically based authentication, integrity and confidentiality services between network devices. Because most VPNs operate at the Internet protocol (IP) layer, they are unaware of, and are transparent to, users, applications, and protocols above the IP layer. In essence, the VPN creates an encrypted IP pipe between network devices for transport layer and applications layer protocols to pass through.

Because VPNs are not aware of anything above the IP layer, they cannot support many of the security requirements of applications, including Web applications. Specifically, VPNs do not

- Provide digital signatures for nonrepudiation of transactions
- Perform content inspection for interpreting the application's data stream for policy enforcement
- Authenticate individual users
- Authorize specific users' access to specific applications.

4.1.3 When to PK-Enable

The types of DoD applications for which the DoD PKI must be used as the primary I&A mechanism include the following:

- Unclassified private Web servers, including those that provide nonsensitive public releasable information
- All email applications

- Web applications that run on unclassified networks.

In addition,

- Web applications on classified networks must use DoD PKI for authentication of the client (browser) to the private Web server;
- Applications that run on unclassified private DoD networks must be interoperable with DoD PKI, and must ensure that users are authenticated using DoD PKI certificates, unless the unclassified network's predominant user community is not required to use DoD PKI certificates for authentication. Such users include retirees, dependents, and academia.

As noted earlier, browser authentication of the Web server should be implemented using DoD PKI server certificates, and server authentication of users and browsers should be implemented using user identity certificates. All I&A and other sensitive data and transactions between the server and browser should be transported over an SSL connection.

The DoD PKI should also be used to provide confidentiality (through encryption), nonrepudiation (through use of digital signatures), and integrity (through use of cryptographic hashes and digital signatures). Discussions of the implementation of PKI-based confidentiality, non-repudiation, and integrity in applications appear later in this document.

Non-Web applications, particularly backend applications that interoperate with frontend Web servers should be analyzed to determine whether those applications would benefit from use of DoD PKI for I&A, confidentiality, integrity, or nonrepudiation.

Documentation of all DoD directives, guidance, and resources for PK-enabling of application-level authentication of users can be found at the DISA PK-Enablement Web site. This Web site should be a primary source of direction and guidance for developers who need to PK-enable their applications. The URL is

<http://www.dodpke.com/sitemap.asp>

Additional documents pertaining to PK-enabling of DoD applications can be found at

<https://iase.disa.mil/documentlib.html#PKIDOCs>

NOTE: For one to access the documentation that is useful for application developers, both sites require submission of a DoD PKI certificate or connection origination from a .mil or .gov domain.

These DoD Web sites present documentation that provides guidance to the developer on implementing and integrating DoD PKI in DoD applications. There are documents that identify the cryptographic standards, protocols, certificate classes and versions, token types, and APIs that must be conformed to.

The sites also provide PK-enabling templates and a knowledge base of DoD PKE lessons learned by other developers.

4.1.4 PK-Enabling Web Applications

The defacto standard for PK-enabling Web applications is the SSL protocol, which, in conjunction with the underlying PKI, provides confidentiality, integrity, and (optional) client and server authentication. SSL was originally designed to be used in applications in which the client has little trust in the server. The server does not need to have much trust in the client. An example is an e-shopping system in which the user wants to be sure that the server is the server he thinks it is before sending it his credit card information, but the server does not care who the user is, as long as the credit card information sent by the user is valid. However, SSL does support an option that enables the server to authenticate the browser based on the user's personal identity certificate before granting the user access to the server.

To date, most Web applications implement only one-way certificate-based authentication. When user authentication is required, these applications use an HTML form to let the user enter a username and static password for identification and authentication.

DoD Web applications, however, must implement two-way mutual certificate-based authentication, that is, browser authentication of the Web server, plus server authentication of users; the first using DoD PKI server certificates, and the second would use user personal identity certificates. Each browser must have its user's identity certificate loaded into it from the user's PKI token (e.g., CAC or FORTEZZA), or from a file on the user's PC hard drive or a floppy disk. Once the user's certificate is loaded into the browser, the browser will be able to pass the user's certificate to the server application during the SSL authentication handshake.

In most cases, SSL is used to accomplish initial session authentication, and then to provide an authenticated, encrypted trusted path over which server authentication of users can be accomplished via LDAP authentication at the application layer (above the SSL layer). The server certificate authenticated by SSL may also be used for LDAP authentication, and DoD PKI LDAP directories can be accessed by LDAP-capable SSL, to retrieve stored certificates and session encryption keys. To establish an SSL connection to the LDAP directory, a port number must be specified to run the SSL service on the LDAP server; port 636 is the standard LDAP SSL socket number for TCP and UDP.

NOTE: SSL traffic may not be able to pass through some application proxy firewalls without first tunneling the SSL protocol to make it transparent to the firewall.

It is unlikely that username and password authentication can be done away with altogether, though it may be possible to limit it to an alternative authentication method, used only when certificate-based authentication fails, or when a user is unable to present a (valid) certificate because he has not yet been issued one, or if the certificate expired when he was on vacation or temporary duty yonder (TDY). Also, certain applications may support non-DoD user communities for whom it is not practical or even advisable to assign DoD PKI certificates. There may be applications in which users require access for

only a limited time (e.g., Coalition applications). Also, in certain tactical applications, lack of direct, reliable access to a PKI or certificate management infrastructure may make the use of certificates impractical.

Like VPNs, SSL does not provide the technology for digital signatures; digital signature capability for Web application users must be implemented through a browser extension or plug-in. Nor is certificate validation (either through retrieval and checking of certificate revocation lists [Curls] or via an online certificate status protocol [COPS] responder) supported by most COTS browsers without the addition of software extensions.

Even if the browser does include the necessary software for certificate validation, a security imperative for all DoD Web applications is that the server applications must not rely on or trust browsers to perform any security tasks. A DoD server application must not rely on a user's browser to validate that user's certificate before sending it to the server. Instead, the server application must perform validation of all user identity certificates it receives.

At this point, the DoD PKI uses Curls for certificate revocation. The local directory used by the server application must be able to store the Curls issued by the DoD PKI certification authority (CA), and the Web server application must be programmed to interpret the directory-stored CURL directly for certificate validation, or the directory and the application must support the use of COPS to enable the application to validate the certificate in real time. The actual certificate validation functionality may be implemented by a Web proxy or Java applet implemented within the server application. Or it may be provided by invoking an external certificate validation system, such as KyberPass's Validation TrustPlatform or CertCo's CertValidator (see Appendix C), most of which include OCSP responders.

In addition to certificate-based I&A, the Web application will also need to include some form of server applet or proxy (augmenting the standard Web server) to perform application-level monitoring and auditing of Web connections, access events, and security violations. Another applet or proxy should be used to implement granular access control, down to the Web page or URL level. When combined with a database (directory) of usernames and permissions (the application's access control list [ACL], this applet and proxy may also perform user authorization for the application.

4.1.4.1 Choosing an SSL 3.0 Tolerant Web Server

There are some COTS Web servers that incorrectly implement the SSL 3.0 specification. This incorrect implementation causes these servers to reject connection attempts from browsers that are compliant with the SSL 3.0 and TLS (SSL 3.1) specifications. Specifically, the SSL 3.0 and TLS (SSL 3.1) specifications both contain the same provision for detection of version rollback attacks. This provision enables the server to detect a particular type of denial of service attack in which a so-called man in the middle captures the browser's SSL connection request in transit to the server and alters that request to reflect a lower protocol version number that is not interoperable with the TLS/SSL version supported by the browser. The server that does not support TLS is supposed to roll back to SSL 3.0 (according to the specified rollback provision), but instead they drop the connection to the browser, resulting in the display of a blank page in the browser.

The following Web servers are known to have this problem and should not be used:

- Domino-Go-Webserver/4.6.2.6 (and possibly later versions)
- IBM_HTTP_Server/1.3.6.3 or earlier (use 1.3.6.4 or later)
- IBM_HTTP_Server/1.3.12.1 or earlier (use 1.3.12.2 or later)
- Java Web Server 2
- OSU/3.2 – DECthreads HTTP server for OpenVM
- Stronghold/2.2
- Infinite Technologies Webmail v. 3.6.1 (use later version) .

4.1.5 PK-Enabling Backend Applications: PKI Toolkits

With the DoD PKI, the PK-enabled applications that do not support SSL can retrieve the same user identity certificates from the DoD PKI LDAP directory as are retrieved by the Web server's SSL authentication service. But whereas SSL provides the basis for Web application security, providing the same security functionality in a non-Web legacy may require use of a proprietary PKI toolkit, such as those sold by Baltimore Technologies, Entrust, RSA, and VeriSign.

PK-enabling of a legacy application often entails extensive changes to the application source code to enable the application to provide direct PKI support and SSL, if desired). This makes PK-enabling of third-party legacy applications, as well as older in-house legacy applications for which source code is no longer available, impossible to PK-enable. In addition, PKI toolkits are generally not available for mainframe and minicomputer legacy applications. In the case of such legacy applications, it may make more sense to implement a security frontend (i.e., an SSO and encryption server) to perform PKI-based authentication and encryption on behalf of the legacy backend. This approach can work securely, however, only if the transmission path between the security frontend and the backend application (or if it communicates directly with the Web server, between that server and the legacy back end) is assured to be secure from external penetration, that is, a VPN would be implemented between the two, to provide a trusted path between them for exchange of sensitive authentication data. Alternately, a firewall may be used to isolate the backend connection and protect it from external penetration.

Be aware that PKI toolkits are vendor specific. RSA Security's BSAFE toolkit, for example, expects the application to use RSA's PKI; Entrust's PKI toolkit presumes use of the Entrust PKI. In some cases, a Java programmer using the Java Software Development Kit (SDK) may be just as effective as a PK-enabling developer using an expensive PK-enabling toolkit to PKI-enable an application. That said, several PK-enabling toolkits have been listed in Appendix C, for those who want to use them.

4.2 IDENTIFICATION AND AUTHENTICATION MECHANISMS

Nonbypassable authentication based on trustworthy credentials should apply to users and to processes, programs, and any other active entity or object that interacts with the application on a user's behalf. In

many cases, applications will not perform their own I&A, but will call out to an external I&A mechanism. This mechanism may be in the underlying operating system, or it may be an external authentication or SSO server.

In certain high-risk environments, a third factor of authentication, in addition to PKI certificate-based I&A, may be desired. This third factor may be a static password, a dynamic password, or a biometric. In some applications, the user's initial authentication may be via a third-factor mechanism, with DoD PKI certificates used as session tokens for all subsequent reauthentications during the authenticated session. PK-enabling of applications to enable them to perform I&A using PKI certificates was discussed in Section 3.1.

There are two ways in which user authentication to the Web application may be implemented:

1. If an SSO has been deployed for the application environment:
 - Obtain the user's I&A information (identity and credentials) via an API between the Web application and the SSO system
 - If you do not wish for the application itself to have to handle user credentials, have the application wait for the SSO system to send a go-ahead for the user, and accept that go-ahead as proof of the user's having been positively authenticated.
2. An alternative is to use PKI-based SSL authentication:
 - Authenticate users based on their DoD PKI or FORTEZZA X.509 Version 3 certificates transferred from their browsers. Cryptokeys would be exchanged via an appropriately strong key exchange algorithm to establish a secure session encrypted using an appropriately strong session encryption algorithm (i.e., the algorithms specified for the DoD PKI or FORTEZZA)
 - Implement an alternate user name and password-based I&A mechanism (e.g., an authentication screen with an HTML form embedded to receive username and password) to accommodate I&A of users who cannot present valid certificates. Do not implement this mechanism in a Java applet; Java applets are too vulnerable to compromise to be trusted to implement a security function such as user I&A.

NOTE: FORTEZZA uses Class 4 certificates. DoD PKI supports both Class 3 and Class 4 certificates, as appropriate, for the sensitivity and mission criticality of the application. All DoD PKI-based applications will eventually transition to Class 4 certificates, according to the schedule for DoD PKI evolution.

4.2.1 Notification of Authentication

Regardless of the I&A method used, when the user is successfully authenticated, before granting the user access to the requested Web application or page, the application needs to present the user with the following information:

- The fact that the user has accessed a government system
- The extent to which this system will protect the user's privacy rights
- The highest classification level of data that may be processed by the application
- The user's responsibilities while using the application to process sensitive or classified information
- The fact that the user's actions while using the application are subject to audit.

In addition, if the application processes classified information, this notification message must also include the following information associated with the user's authenticated username:

- Date, time, and origination (client IP address or domain name) of the most recent previous authentication of this username
- Number of unsuccessful I&A attempts from this username subsequent to the most recent previous successful authentication.

The best way to implement the required warning message would be to write a script that can dynamically generate the HTML for the warning message and present that message:

- In the main browser window, with a clickable link (text or labeled icon, or both) at the bottom of the screen, enabling the user to acknowledge having read the notification message; clicking this link would automatically launch the application session requested by the user
- In a separate pop-up browser window, with a clickable link (text or labeled icon, or both) at the bottom of the window, enabling the user to acknowledge having read the notification message; clicking this link would then automatically close the pop-up window. You may chose to code the window in a way that prevents the user from closing the window via the standard button at the top of the window, that is, make the user's explicit acknowledgment of the notification window the only way the user can close the window
- In a banner at the top of the current Web page (e.g., coded with attention-grabbing effects such as a bright colors, animation, etc.), with a clickable link (text or labeled icon, or both) within the banner, enabling the user to acknowledge having read the notification; clicking this link would then automatically make the banner disappear.

4.2.2 Client (Browser)-to-Server Trusted Path

Because most operating systems do not establish trusted paths for exchange of authentication credentials and other sensitive data between clients (browsers) and servers, some other means must be used to ensure that sensitive security information, such as passwords, are sent securely between them. In non-Web systems, this usually means client encryption of the password before transmitting it; it may also mean implementation of a VPN to encrypt the network connection over which the password is sent.

In Web applications, a trusted path for password transmission is simulated through the use of HTTPS to transmit HTML authentication forms over SSL-encrypted connections between clients and servers. We say simulated because implementation of a true trusted path is virtually impossible in Web applications. There are a number of well-documented attacks used to fool users of Web browsers into thinking they are connected to a different URL or server than the one to which they are actually connected. These attacks include Web spoofing through use of changed URLs, rare URL syntax, or URLs very similar to valid URLs, with various techniques used to obfuscate the actual URL being visited. In all such cases, use of an SSL connection will not prevent the problem because the bogus Web server can use HTTPS and establish an SSL connection to the user's browser just as easily as a valid Web server can.

There is little that a Web application developer can do to prevent these exploits, except to encourage the use of domain names and subsidiary URLs that are very simple and unlikely to be misspelled or forgotten by the user, and to purchase other domain names similar to the valid domain name to be used by the Web server. For example, if the Web server's domain name is validname.mil, the organization may want to also purchase validname.gov, validname.org, and validname.com to prevent these names from being used by attackers to set up bogus Web sites to mimic the true validserver.mil site. Before selecting a domain name, the organization should search for all similar domain names to determine whether they may be owned by potential attackers intending to host malicious Websites.

Applications should be written to trust only information (authentication information, configuration information, input data, query results) received over trustworthy channels. No unauthenticated transmission should be considered trustworthy. TCP/IP authentication should not be trusted as the sole authentication mechanism. To implement a trusted channel, take these measures:

- E-mail messages and files sent over untrusted channels should be digitally signed to prevent them from being modified or forged.
- Hidden fields should be digitally signed or encrypted, or both (using a key only the server can decrypt), by the browser before they are returned to the server, which should then validate the digital signature.
- Cookies returned by the browser to the server should not be considered trustworthy unless transmitted over an SSL/TLS encrypted and authenticated channel.

- Files and data containing references to other data, particularly remotely stored data, (Examples of remotely stored data are HTML or XML that includes references to files such as style sheets and document type definitions [DTDs]) should not be trusted unless the file containing the reference has been received over a trusted channel and/or digitally signed or encrypted. This measure will help minimize the possibility of an attacker tampering with the data reference. To tamper might be to substitute a different style sheet for a Web page that whites out critical words, inserts new text, or otherwise defaces the Web page. Similarly, external DTDs could be modified to make the document unusable (by adding declarations that break validation) or to insert different text into the document.

4.2.2.1 Extending the Chain of Trust to a Backend Server

In the case of a Web application in which the Web server interoperates with a backend database or other backend server, the chain of trust established between browser and Web server must be extended to the backend database server. The Web front end in this scenario authenticates the browser (and its user) based on X.509 certificates. The question is how to convey that user's credentials to a backend server that expects to see a username and password, and how to extend individual user accountability beyond the Web front end to the backend database.

The LDAP (or X.500) directory in which the users' X.509 certificates are stored can also be used to map each certificate to the username and password that must be transmitted to the database back end. The application front end would simply issue an LDAP request to that directory, conveying the user's personal identity certificate, and would retrieve the corresponding username and password for transmittal to the database back end.

One approach would be to create a pool of connections from the Web front end to the backend database, with one connection allocated per security role supported by the application, in essence extending RBAC to the backend server. If individual accountability is required by the backend server, the Web front end would have to create a separate connection for each user to the database back end.

Another approach to this problem would be to use the Java J2EE application server platform. The Java Authentication and Authorization Service (JAAS), supported by the J2EE, was developed to address these issues, that is, authentication of users of Java applications, applets, beans, and servlets, and authorization of users to ensure they have the access control rights (permissions) required to perform the actions they request and attempt.

JAAS implements a Java pluggable authentication module that references the Java server's configuration to determine which authentication technology, or *LoginModule*, should be used (e.g., username and password, PKI, or biometric). The credentials used by the user to log in to the J2EE server can be passed by the server to a backend database for authentication, or other credentials may be used to log in the user to the backend server. Connection and authentication between the J2EE server and the backend database is handled by the Java Database Connectivity API/service (JDBC).

4.2.3 PKI-Based I&A

As noted in Section 4.1.4, I&A in DoD Web server applications must be PK-enabled to validate and authenticate users based on their DoD PKI X.509 identity certificates. Some COTS X.509 certificate validation software products that may be integrated with Web server applications are listed in Appendix C.

As also noted in 4.1.4, DoD Web applications, although using certificate-based I&A as their main I&A mechanism, should also support username and password authentication to accommodate users who cannot authenticate using DoD PKI certificates (due to lack of a valid certificate). Implementation of Web-based username and password I&A over SSL connections is discussed in Section 4.2.

NOTE: The classes of DoD PKI certificates to be used for PKI-based I&A on DoD public, private, mission critical, and classified DoD Web servers are specified in ASD C3I Memorandum, Public Key Enabling (PKE) of Applications, Web Servers, and Networks for the Department of Defense (DoD), 17 May 2001 (DoD PKE Policy).

4.2.3.1 Browser Use of Hardware Tokens

On the client side, the browser must be able to accommodate the use of a hardware token as the storage device for the user's personal identity certificates and private cryptokeys. This means that the browser must provide an interface to the hardware token.

To use the public key capabilities of the CAC to enable the user to establish an SSL connection with the Web server for client authentication, the browser needs to be able to access the data and functions on the CAC. There is no single standard method for doing this. Netscape Navigator uses PKCS#11, also known as Cryptoki, from RSA laboratories as its card access protocol, whereas Microsoft's Internet Explorer uses Microsoft's own proprietary CryptoAPI. Both APIs have some similarities, such as allowing a Dynamic Link Library (DLL) plug-in to the browser to provide the card interface service. These browser plug-ins should be obtained from the vendor of the smart card reader being used with the CAC.

If FORTEZZA is the token being used, Microsoft Internet Explorer 5 (and later versions) includes a FORTEZZA cryptographic service provider (CSP) plug-in. Netscape Communicator's Cryptoki interface also supports access from the browser to the FORTEZZA card.

4.2.4 Reusable (Static) Password I&A

Although DoD Web applications must use DoD PKI as their primary I&A mechanism, for most applications, reusable (static) username and password authentication must also be implemented. Specifically, username and password authentication may:

- Provide an alternative authentication mechanism for users who are unable to present PKI certificates to the server. This is particularly important in the short term, for not all DoD users have yet been issued DoD PKI identity certificates
- Act as a third factor of authentication in applications where PKI authentication alone is not considered sufficiently robust or trustworthy, and use of dynamic passwords or biometrics is not considered a practical alternative
- Serve as a preliminary application authentication mechanism in applications for which certificate use is impractical, such as, coalition applications that support dynamically changing non-DoD user communities.

In addition, username and password authentication is likely to continue as the predominant I&A mechanism in non-Web applications, including backend database applications and other legacy applications that may need to interoperate with DoD Web applications.

4.2.4.1 Implementing Reusable Password I&A in Web Applications

Username and password I&A is being implemented increasingly through use of SSO systems that provide a common, centralized authentication service to multiple applications, which instead of implementing I&A themselves, receive the authenticated credentials of users via an API to the SSO system. A discussion of application integration with SSO systems appears in Section 4.2.3.

When implementing static password authentication, do not use the Web Server's basic authentication capability without also implementing SSL/TLS encryption between the browser and the server, because basic authentication does not encrypt passwords before transferring them. Although HTTP digest authentication, in which the password is encrypted, has been specified as an alternative to basic authentication, in reality HTTP digest authentication is often implemented incorrectly by, or incompatibly between, different brand names of browsers and Web servers. Therefore, SSL/TLS is the encryption mechanism that should be used between browsers and servers for transmission of authentication and other sensitive information.

Furthermore, never use unencrypted or persistent cookies to store and convey authentication data. Such cookies are not stored or handled securely by the browser and thus are easily captured and replayed or otherwise compromised.

The correct way to implement username and password I&A in a Web application is through the use of either one-time encrypted cookies, or through the use of encrypted HTML hidden form fields. Here are ways that the latter is used:

1. Before granting the user access to a requested Web page, the application should present a pop-up HTML form with two blank fields: the first prompting entry of the user's username and the second prompting entry of the user's password.

2. As the user types his or her password into the HTML form field, that password should not be echoed back in cleartext (where it could be seen or copied by a shoulder surfer). Instead, the HTML field should echo back a series of nonmeaningful characters (asterisks are most often used) corresponding in quantity to the number of characters in the user's password.
3. The HTML form containing the user's password should be transmitted over an SSL/TLS encrypted link. This transmission should be instigated by an HTTPS *POST* operation, not by an HTTPS *GET* operation. This is because HTTP *GET* writes hidden HTML form field data (passwords) in cleartext in the browser's history log, where it can easily be discovered and copied by a malicious user who gains access to the authorized user's system in that user's absence.

4.2.4.1.1 Password Digests Instead of Passwords

Rather than sending the user's whole password over the SSL/TLS-encrypted link, it is more secure to calculate and send a digest (cryptographic hash) of that password, as the password digest is far less susceptible to capture and replay attacks. In either case, the hash mechanism used must be National Information Assurance Partnership (NIAP) CC validated.

4.2.4.2 Confidentiality and Integrity of Usernames and Passwords

To ensure protection of sensitive I&A data (usernames and passwords) and, indeed, all sensitive data, develop applications according to the following guidelines:

- *Use HTTPS with SSL/TLS for all sensitive data transmissions:* As already noted, any time a static password or password digest (or any other sensitive user input) is to be transmitted between browser and server, the connection over which the password and digest is sent should be encrypted with SSL/TLS. Indeed, it may be desirable to use HTTPS rather than HTTP, which means that SSL/TLS will be invoked for every browser-server transaction, and not just for the initial I&A transaction, to ensure that the password is never inadvertently retransmitted in cleartext during a session reauthentication, or in an HTTP referrer field to another Web site.
- *Use only PKI certificates as session tokens.* The browser should not use the password and digest embedded in an HTML form field, in a URL, or in a cookie as the user's session reauthentication token. Only DoD PKI certificates should be used as session tokens for all session reauthentications.
- *Never store or send sensitive data in a cookie.* Cookies should not be used to store any kind of sensitive information. Cookies used in a Web application should always conform to the application's security policy, specifically with regards to whether the cookies used are persistent or nonpersistent or secure or nonsecure. If the system on which the browser runs does not provide adequate security, the application should use only nonpersistent cookies. If SSL/TLS is used for transmitting cookies, the application should be consistent and use only secure cookies.

- Use *HTTP POST*, not *HTTP GET*. Also, remember that passwords (and other sensitive data) entered by users into HTML form fields may be stored in cleartext in the browser's history log or a proxy log unless *HTTP POST* rather than *HTTP GET* is used when submitting the HTML form. See 4.4.3.2 for more information.

4.2.4.3 Unsuccessful Log-In Attempts

Applications that support username and password authentication should be configurable to audit every unsuccessful log-in attempt (correctly entered passwords should not be recorded in the audit log). In addition, the authentication mechanism should be configured to allow only a finite number of log-in attempts, after which the application should send an alarm to the administrator notifying him of the attempted security violation; the application may also be configurable to automatically impose a lockout period of a configurable duration on the username from which the unsuccessful log-in attempts originated. This will prevent the hacker from further abusing that username to launch a password-guessing attack.

Configuration of log-in thresholds, lockout periods, and auditing of unsuccessful log-in attempts are standard features of system-level I&A mechanisms, SSO systems, and authentication middleware. If the application's I&A mechanism is being developed from scratch, not using one of those features, you must be sure it implements the necessary administrator-configurable log-in attempt threshold, auditing, and configurable disposition (configurable lockout period or administrator alarm, or both).

4.2.4.4 Explicit Log-Out

Web applications that require users to explicitly log in (versus transparently authenticating users via PKI certificates alone) should also provide a mechanism to enable those users to explicitly log out again. When the user logs out, the application should purge from memory and cache all initial authentication data, session reauthentication tokens, and session IDs associated with the user. If the user fails to log out, a session expiration time-out should be enforced by the application, at which time all authentication and session information associated with the user should be purged, and the user, if he attempts to reconnect to the server, should be required to reauthenticate to establish a new session.

4.2.4.5 Password Management

The password management system used by the application must enable the administrator to configure password expiration thresholds, and it must require users to change expired passwords before they can access the system. The application's password management system must also enforce strong password rules on passwords selected by users. It must prevent users from selecting weak passwords and provide them with guidance on how to construct strong passwords. Guidance for users could be an informational message on the HTML form used for password selection, reinforced by a returned error message or a pop-up window triggered by a user's attempt to select a weak password. (See Section 4.1, Requirement #4.1.19, of the *Recommended Standard Application Security Requirements* document for information on strong password construction).

The application must not predetermine or require the use or assignment of particular usernames (other than that associated with the Web server's "nobody" account), and it must not prevent the administrator from defining all usernames, including any usernames assigned to application processes (rather than human users) and initial passwords. The application must not require use of any anonymous accounts, other than the Web server's nobody account, which should be assigned a username other than "nobody" if at all possible.

4.2.5 Single Sign-On Systems

Use of an SSO solution in a Web application context is intended to eliminate the need for users to reauthenticate to Web applications when they access more than one application via a Web portal during the same session. Ideally, the user will have to provide his credentials (username and password, certificate, etc.) only once: to access the Web portal the user and will not be required to reauthenticate to the individual Web applications made accessible by the portal.

SSO comes in two main forms: desktop SSO systems, such as the Kerberos-based Windows 2000 Active Directory Domain, and Web access management SSO systems, such as RSA ClearTrust and Oblix NetPoint (cookie based).

One of most established, popular SSO frameworks is Kerberos, which is integrated into both Solaris and Windows 2000. Kerberos may also be found as an enabling component in SSO systems that use PKI certificates rather than Kerberos tokens as their end-user authentication credentials. One example is the Kerberos-encrypted password store technology used with an LDAP Directory and X.509 certificates in the Navy's TFWeb SSO implementation, to ensure that the original passwords cannot be recovered. Kerberos has also been implemented with the SecureID one-time password system, as in the DoD High Performance Computing Modernization Program.

To use the authentication service of most SSO solutions, server applications must be aware of the fact that they operate within an SSO framework. Integration of SSO code into clients and servers alike is required, which means modification of existing clients and servers if they are to be incorporated into the new SSO framework. An exception is the server that already has a clear challenge-response sequence that can be automated, in which case required SSO modifications will be limited to the client. These modifications will enable the client to obtain the user's authentication credentials not directly from the user, but from the SSO framework.

The majority of current SSO frameworks use either username and password or X.509 certificate-based authentication (with the SSO system performing the certificate validation). The SSO system handles identity and rights management for the entire operating environment, including all of the applications and operating systems in that environment. The SSO system protects the user authentication and authorization information by transmitting it between entities on the network via SSL-encrypted links. Once the SSO framework is in place, access to the back end is possible only through the SSO server.

The Web access management SSO system most likely uses encrypted temporary session cookies to pass authenticated user identity information from browser to Web server. Note that use of cookies for

this purpose is not a good option for Web applications that have to authenticate across domain boundaries: for security reasons, the HTTP cookie specification allows a Web server to manipulate only cookies that will be sent back to itself or to other servers in the same domain. This means that whereas a cookie sent by one server will be sent to other servers in the same domain, it will not be sent to servers in any other domain. However, multidomain SSO systems are available that use proprietary Web server plug-ins to enable authentication securely across domain boundaries (e.g., Microsoft's Passport SSO). In the SAML, being defined by the OASIS, promises to provide a standard XML-based security mechanism for exchanging authentication and authorization information between domains.

Web-based SSO systems may also provide an alternative mechanism for passing authenticated user identity data via HTTP request header fields. HTTP headers alone are not trustworthy; they can be intercepted and spoofed. SSO of Web applications using HTTP request headers without a secure authentication token (e.g., encrypted cookie, signed assertion) cannot be trusted. Web applications that sit behind a Web portal that uses HTTP header-based identity transfer should reauthenticate users who connect to them. Implementation of an SSO agent on the Web server running the application will enable passing the SSO secure authentication token. The advantage of using an SSO Web server agent is that it tightly integrates the Web application with the Web portal's SSO system. However, some organizational firewall policies prevent the transfer of SSO data between Web server agents and the central SSO system.

Another difficult aspect of implementing an SSO framework is defining the connectors between Web portals and backend legacy systems, such as database servers. Connectors are the executives for SSL-enabled mutual-authentication sessions between servers. To date, there seem to be few efforts under way by major SSO or Web software vendors to develop connectors for their systems.

NOTE: Aside from restrictive firewall policies, firewalls can cause other problems when SSO systems are deployed. SSO generates numerous network processes—more than can be efficiently routed to port 443. This means that some SSO vendors use other ports that need to be opened to accommodate the SSO traffic. If the firewall's ports-and-protocols policy—or a general lack of coordination between firewall port assignments—prevents use of other port numbers by the SSO traffic, SSO transfers may not be able to cross firewall boundaries. Though this is mainly a problem for infrastructure security and networking teams, application developers should be aware of it insofar as they influence the organization's decision to use SSO rather than a more firewall friendly authentication scheme in the Web application environment.

4.2.5.1 Security Service APIs

Security service application programming interfaces are necessary in SSO frameworks. The most common standard security service application program interface is the GSS-API, which is used to enable application calls to a range of underlying authentication mechanisms and technologies. Three SSO schemes—Kerberos v.5, the OpenGroup (formerly Open Software Foundation) Distributed

Computing Environment (DCE), and the European SESAME scheme—all use GSS-API as a programming interface to enable application use of SSO I&A capabilities.

SSO solutions based on GSS-API most often base authentication on symmetric cryptographic tokens transferred from the SSO to a cache maintained by the operating system's security service provider. These tokens can either directly provide access to services or may act as time-limited authenticators that in turn are used to obtain one-time service access privileges.

4.2.5.2 SSPI in Windows NT and Windows 2000

For distributed applications (including Web applications) on Windows NT and Windows 2000 systems, the API to the operating system authentication services is the Security Support Providers Interface (SSPI), which sits between the application layer network protocol (e.g., HTTP, FTP) and the Windows Security Support Providers (i.e., NTLM, Kerberos [which is now supported in Windows 2000], SChannel [the Windows implementation of SSL/TLS], and distributed password authentication [DPA]).

As an alternative to SSPI, Windows operating systems also provide the graphical identification and authentication (GINA) API, which is intended to allow vendors and other developers of alternative authentication mechanisms (i.e., alternatives to username/password authentication) and pluggable authentication modules (PAMs) to implement those systems on Windows. Also noteworthy is the cryptographic application programmatic interface (CAPI) to the underlying cryptographic services provided by the operating system and used in conjunction with any cryptography-based authentication scheme.

Unlike systems based on GSS-API, Windows NT/2000 does not retrieve a set of credentials that match the authentication. Instead, Windows encodes the authentication response and forwards it to a central repository known as a *domain controller*. If the authentication response is correct, the controller designates the client as authenticated for access to its security domain. The different services in the domain accept this designation by the domain controller and allow the controller also to make all authorization decisions related to the authenticated client.

4.2.5.3 Vulnerabilities of SSO Systems

Whereas in traditional Web authentication each Web server is responsible for safeguarding the authentication information of its users, when an SSO system is used, those data are centralized in one SSO server. Thus, compromise of this central server would be devastating to the security of the entire Web infrastructure that relies on it.

In addition to authentication data, the SSO server often maintains user profile information on all registered users, also storing this information centrally and making the server an extremely attractive target for attack, both for denial of service and unauthorized access.

The centralized service model implemented by SSO is antithetical to the distributed nature of Web-based computing. Indeed, distribution is what gives Web computing its robustness and popularity.

The effects of a denial of service attack on an SSO server are acute. Obviously, the usefulness of the SSO system increases in direct proportion to the number of Web applications and other systems that use it. But as the number of systems that support SSO grows, the effects of an outage, intentional or accidental, also grows. An attacker might accomplish such a denial of service by flooding an SSO-supporting Web site with bogus user log-in requests.

The usual approach to service availability is to replicate the vulnerable service sufficiently to make catastrophic failure unlikely. However, because older SSO systems (e.g., ClearTrust 4.6) do not implement robust directories, there is a concern about how such systems can handle the fundamental problems of key distribution and database replication on a large scale. Replicating the service would require multiple copies of private keys across the replicated SSO servers, thus increasing the exposure of those keys to potential compromise. Newer SSO systems, by contrast, are based on directories with clear replication architectures.

A specific denial of service attack exists in SSO systems that store their SSO tokens as cookies in users' browsers. An attacker could impersonate the SSO server and delete the token cookies from the browsers at will.

4.2.6 Other I&A Technologies

4.2.6.1 One-Time (Dynamic) Password Systems

Applications that base authentication on reusable (static) passwords are subject to numerous attacks that target the inherent vulnerability of those passwords, such as password-guessing attacks, password capture and replay, shoulder surfing, and other forms of social engineering. In environments where such attacks are likely, it makes sense to replace reusable passwords with one-time (dynamic) passwords.

One-time password systems are system-level modules that must be integrated into the underlying operating system. These systems provide APIs through which the application's I&A process can invoke the one-time password system to validate the one-time password sent to the application by the user. The user obtains the password from a handheld token or a software program on his client.

It may also be desirable, in a distributed multiserver or multidomain application environment, to use a one-time password system in combination with a distributed authentication system, such as X.509 or Kerberos. In the DoD's High Performance Computing Modernization Program, for example, RSA Security's SecureID one-time password system is used for preliminary user authentication. The system assigns the user a Kerberos token for transparent interprocess reauthentication throughout the distributed system.

COTS and public domain one-time password systems are listed in Appendix C.

4.2.6.2 Biometric Authentication Systems

Although DoD Web applications must use DoD PKI as their primary I&A mechanism, as noted earlier, there may be applications for which a third factor of authentication is desired. In a tactical environment,

for example, the risk of compromise may be higher due to the physical exposure of the computing environment.

Biometrics is becoming an increasingly viable as an alternative to usernames and static or dynamic passwords, as a third authentication factor in Web applications (or a primary authentication factor in non-Web applications). Used in conjunction with DoD PKI, biometrics can improve the security of the PKI private keys. A biometric would be used to authenticate a user before granting that user access to his or her private key stored on the same token (CAC) as is the biometric, or on the user's PC hard drive.

Please note that although biometrics is one of the advanced security technologies (along with VPNs) whose use is encouraged by the Office of the Secretary of Defense, definitive DoD policy on use of biometrics has not yet been approved. A draft policy is only now being circulated for review. Once approved, this policy will become part of the DoD Directive 8500 (GIG Information Assurance [IA]) series. In addition, the CC medium assurance protection profile for biometric products currently exists only as a draft under review.

If use of biometrics in an application is approved by the application's owner and accreditor, the developer should strive to use a biometric COTS product that has been approved by NSA, at least until there is a CC protection profile and NIAP certification process for biometric products. To date, only biometric products from Bioscrypt have been approved by NSA or certified (low assurance) under the CC. These Bioscrypt products, and other biometric products in use within DoD, are listed in Appendix C.

4.2.7 Pluggable Authentication Modules

A PAM provides a practical method for integrating an authentication service into non-Web backend and legacy applications. The PAM acts as an API layer between the application and authentication middleware or the operating system authentication service. The PAM obscures the implementation details of the authentication mechanism, presenting the application developer with a single, consistent API from the application regardless of the actual authentication mechanism being used. The PAM manages the specific, unique interface calls to the underlying authentication mechanism, determining at run time which authentication mechanism to invoke by checking the configuration set up by the local system administrator. Thus, the PAM enables the authentication mechanism used by the application to be changed through a simple configuration parameter, without the application itself having to be rewritten.

Many UNIX (and Linux) systems, including Solaris, support PAMs. To date, Windows does not support PAMs, though at least one prototype PAM for Windows, developed by academia, is available as open source software; this is listed in Appendix C. In addition, Appendix C lists several add-on PAM implementations for UNIX systems. Application developers working on UNIX systems should also refer to the UNIX *man* page on PAMs for the particular version of UNIX on which they are developing the application.

4.3 AUTHORIZATION AND ACCESS CONTROL MECHANISMS

In a distributed Web application environment, it may make sense to implement authorization centrally using a third-party authorization server such as Hewlett-Packard's Praesidium Authorization Server (see Appendix C). Such a server can be as a central authorization rules engine to define and enforce access rules for multiple Web and non-Web applications within the operating environment. Application authorization then becomes part of the operational infrastructure, and it is performed consistently across applications, instead of having to be implemented separately for each application.

If the Web application must perform authorizations itself, its process for allocating authorizations must be correct nonsubvertible, though it will have to trust the authentication system to tell that entities and objects to be authorized have been properly authenticated.

The authorization (privilege) information used by the application may be stored and managed in the existing directory (LDAP, X.500) used by the PKI to store the user's X.509 certificates. The retrievals—via LDAP (the application's authorization function will have to be LDAP enabled—of this authorization information from the directory by the application should be secured via SSL when the data are in transit. The data at rest will be protected by the directory's access controls and other security measures.

If standard user/group/world authorizations are inadequate, the application's authorization system may have to implement ACLs and role-based authorizations, with finer-grained access controls (e.g., role-based access control) implemented to support them. In messaging applications, role-based addressing may be used to implement this granularity.

Because of the additional requirements for access control of classified, sensitive, and Mission Category I unclassified data, it may be necessary for the application to augment the underlying operating system access controls by ensuring that such data at rest are encrypted when they are stored on the Web server (or application server). See Section 4.3.7.

4.3.1 Implementing a Single Application Entry Point

Most Web applications incorporate several component applications/systems besides the Web server, HTML pages, and CGI scripts, and others that make up the core of the Web application. Examples are an FTP (file transfer protocol) servers, SMTP (simple mail transfer protocol) servers, POP (post office protocol) servers, news servers, database management systems (DBMS), directory servers, and proxy servers. Often these components are distributed across multiple physical systems

Given its multicomponent capability, a Web portal can better protect the Web application and the data it serves and processes. When the portal is implemented, the Web server application only allows one possible way for users to invoke or access these component programs, scripts, and data stores. For example, write a Web portal application that provides the single entry point through which users can launch the application's component programs, issue queries to its database, and take other actions. Within this Web portal, a checkpoint program will make the access decisions on behalf of the

application, deciding, based on a user's privileges, which of the application's component programs and databases that user may access. By centralizing all user access to a single entry point, the Web portal will also make application-level auditing easier to implement.

The Web portal should be the only way for users to access any application components on backend servers. Firewalls should be implemented between the Web server and the backend servers to prevent users from directly connecting to those servers. Access controls on the backend servers should be configured to prevent users from directly accessing the backend applications. All of the backend servers, as well as all Web portal nonuser processes that directly invoke or access those components, should be protected from user access by setting their privileges higher than user. Finally, any backend applications that have been Web enabled should have their older, non-Web user interface software modified to prevent user access via any interface other than the Web interface. It is not possible to prevent truly determined users from accessing the backend servers directly from outside the Web portal, but these measures will certainly inhibit the frequency of such direct accesses.

Although it provides the single access method for the users, the Web portal must not impede the successful execution of processes within the Web application, nor trusted processes external to it, that need to invoke component programs and access component data stores. For external processes, this trust should be established through interprocess authentication (see Section 5.7). It may also be desirable to implement interprocess authentication of the Web application's own processes before allowing them access to highly sensitive data stores, programs, or scripts.

As noted before, the portal cannot and should not be expected to prevent determined users from accessing and launching component programs from outside of the application. The portal itself is an application, and contains no trusted computing base (TCB) or reference monitor with which to authoritatively enforce access restrictions. That is why it is critical that the underlying operating system's and DBMS's access controls be configured correctly and restrictively as possible to strictly control direct user access to the programs and data protected by them.

Nevertheless, with its self-contained checkpoint program, the Web portal, acting as the application's single entry point, can control which interfaces to its components are presented to users. In this way, the Web portal's graphical user interface (GUI) presentation to the user can be used to reinforce (versus enforce) the Web server's and backend servers' underlying access controls by presenting to the user only the interfaces (e.g., Web links) to programs and data stores the user is privileged to access.

NOTE: Strict access control must be implemented on each file (Web page, CGI script, etc.) accessible via a URL, regardless of how that URL is input by the user (typed in, bookmarked, or clicked link), and regardless of whether there is a Web portal.

4.3.1.1 Implementing the Web Portal's Checkpoint Program

The Web portal's self-contained checkpoint program will make the user-access-to-application-components decisions on behalf of the Web portal. In essence it will determine for the portal which

Web links should be presented to a particular user, based on that user's authenticated identity, and possibly the user's role, and the authorized access privileges associated with that identity or role, or both. But before the checkpoint program can make these decisions, the user must be positively authenticated to it, either via an API to an SSO server, or via SSL (with an authentication screen implementing an HTML username and password entry form as a backup).

In addition to performing user authentication, the checkpoint program should manage users' authorizations and track any security exceptions and violations that occur during the user-application session. It should decide what action should be taken in response to an exception and violation based on its severity. The checkpoint program may associate a username with a specific role or user group and grant the user's authorizations (access privileges) based on that role or group, instead of or in addition to the user's individual identity. In this way, the Web portal and checkpoint program can be used to implement a type of application-level RBAC in systems where RBAC is not provided at the operating system level.

The checkpoint program may also control and perform other application-level security checks, including session inactivity time-outs, and countermeasures to session hijacking and spoofing. The checkpoint program's security policy will determine which security checks are performed and in what order, and how to handle failures of security checks.

Implement the checkpoint program to be easy to integrate, portable, and extensible, enabling the same checkpoint code to be reused in other Web applications with minimal modification. To do this, encapsulate the application-specific aspects within the checkpoint's security policy, which should be linked into the checkpoint program as a run-time configuration library, rather than embedded into the program itself. This will make it possible for the same Web application to be used by different organizations with different security policies.

Use of the checkpoint will make it easier for you to turn the application's security controls off and on as necessary for debugging and unit testing in the development environment. It will also enable you to experiment with a Web application that may be used by different organizations. You can observe its operation under the constraints of different security policies. Each organization's true operational security policy can be turned on when the application is ready for operational testing and deployment.

4.3.1.2 Limitations of and Alternatives to Web Portal Security

Web portal pages must not be relied on as access control mechanisms. They do not provide the necessary assurance that a true access control mechanism requires. Just because a Web link is not present on the portal page does not mean it cannot be discovered (through guessing of the URL) and typed into the browser's "Go to" or "Location" line. Even if directly typed URLs are rejected through input validation, attempting to simulate access control through use of Web portal links and input validation is a form of security through obscurity. Web pages not linked on a portal page can still be discovered and accessed by determined users. For this reason, all Web resources behind the portal page must be appropriately protected by the underlying operating system access controls.

As mentioned, the Web server application can be programmed to reject any URLs the user submits by directly typing in the browser's "Go to" or "Location" line, or by clicking on a stored bookmark, particularly if those URLs are not also coded as links in the portal page. If a URL is rejected, the application should return an error message to the user, warning him that he will be automatically redirected to the Web portal page, and the application should then perform this automatic redirect within a few seconds of returning the error message. By forcing the user to enter the Web site via the portal page, the application reinforces the message that the user can access Web application resources only after being authenticated by the portal and by clicking those resources' corresponding links on the portal page.

However, just as excluding links from the portal page does not constitute true access control, neither does input validation and rejection of a typed-in or bookmarked entry. As stated before, determined users will find ways to bypass the Web portal altogether to access restricted resources, so access control must be applied to the resources themselves (Web page files, databases, scripts, etc.) to truly protect those resources from direct user access.

4.3.1.2.1 Funneling Users Through a Web Portal May Not Always Make Sense

In some applications, there may be no good security reason for requiring users to access links on the Web site only via a Web portal. For example, a Web application that does nothing more than serve unclassified data in static HTML pages or portable document format (PDF) downloads may not need to control its users as strictly as does a Web application that grants its users access to application processes that can be used to read sensitive or classified data, or submit or manipulate data. I&A and authorization may still be required, but strict funneling of users through a single entry point may not be.

When designing the Web application, determine whether there is a security imperative (e.g., a security policy directive) to prevent users from typing in URLs that are included as links on the Web portal page or anywhere in the Web site. Instead of implementing I&A and authorization in a way that requires users to be funneled through a Web portal page, invoke the necessary I&A and authorization on a per page/per URL basis. Implemented this way, the Web application will not deprive the uses of the convenience of being able to bookmark frequently visited Web pages or force them to navigate their way back to those pages via a main Web portal page each time they need to return to them.

If a Web portal is determined to be a desirable convenience but not a necessary security measure, write the application so that user I&A and authorization processes are invoked for each Web page requested via a valid URL submitted by the user, regardless of the user's point of origin. For example, the user's browser may be displaying another Web site altogether or a blank browser screen when the user submits the URL. Valid URLs may be those URLs that do appear on the Web application's portal page, with input validation rejecting any others. Or it may include all URLs accessible from links on any of the Web pages behind the Web portal. In short, valid URLs that are considered safe may be directly typed in or accessed via bookmarks, with the simple act of requesting the URL triggering the I&A and authorization to access the requested page. There would be no requirement that the user must first visit the Web portal page to be authenticated and authorized.

4.3.1.3 Distributed Access Management Systems

Web access management systems, such as RSA ClearTrust, are one way to implement this kind of checkpoint program functionality, that is to centrally control and manage user access privileges to Web-based resources based on defined user attributes (e.g., roles), security policies, and other business rules. The access management system enforces the Web application's access control policy consistently across all of the application's distributed components.

Distributed access management systems generally provide the following security services to the distributed Web environment:

- SSO
- Authentication credential management, with LDAP data abstraction layer (DAL) interoperability to existing credential data stores/directories
- User privilege management, authorization, and delegation management
- Rules-based access control and security policy enforcement
- End-to-end auditing and reporting.

See Appendix C for information on RSA ClearTrust and other Web access management systems.

4.3.2 Interoperation with System-Level Access Controls

4.3.2.1 Web Server Access Controls

Most COTS Web servers enable access control for individual Web pages. For example, Netscape's Web server can control individual page access either through use of an .htaccess file in the file system directory or through special directives that can be configured in the server's access control administration page, which centralizes all access control information in a single location, instead of distributing it across multiple .htaccess files in multiple file system directories. Similarly, a centralized access control administration page provides only a single target for attackers.

Note that access control of the Web server's files, including Web application executables, Web pages, scripts, and others are ultimately enforced not by the Web server application, but by the underlying operating system's file system access controls. These access controls must be configured correctly to protect the files and the Web server executable itself, so that the Web server is authorized appropriately at the operating system level to access and retrieve the HTML pages, CGI scripts and others it must serve to users. Moreover, these access controls should be configured—and the Web server processes should be assigned privileges—in such a way as to prevent users from directly accessing these resources in the file system if they manage to bypass the Web server.

Please refer to the DISA Field Security Operations Web Application Security Technical Implementation Guide (30 September 1998) for more information on the configuration requirements for DoD Web server access controls.

4.3.2.1.1 Dead Pathnames and URLs

Application responses to user submissions of nonexistent file pathnames or URLs should not reveal any information that could help an attacker understand the directory structure of the file system. Instead of returning an HTTP 404 error, the application should be programmed to redirect users who enter a dead pathname or URL to a central site map or index page. Some Web servers and application servers that serve dynamic pages automatically do this.

4.3.2.1.2 Relative and Truncated Pathnames and URLs Entered by Users

Reject all relative pathnames and truncated pathnames/URLs entered by users. Present an error message to the user who enters such a pathname or URL or automatically redirect him to the main Web page or portal page (e.g., index.html or main.html). Or do both.

4.3.2.2 Database Management System Access Controls

Database access control should be performed by the DBMS only in conjunction with the underlying operating system file system. Do not use database views as an access control mechanism. For one thing, this is not their purpose. More importantly, they do not provide the assurance that access control requires. Attempting to simulate access control through the use of views is, at best, a form of security through obscurity. Security through obscurity is both undesirable and insecure. Data not seen by a user in a view can easily be discovered and accessed if those data are not also appropriately protected by the DBMS and operating system access controls.

Please refer to the DISA Field Security Operations Database Security Technical Implementation Guide (30 October 1999) for more information on the configuration requirements for DoD Web server access controls.

4.3.2.2.1 Time and Date Stamps and Data Access Notification

In the case of a DBMS, time and date stamps will automatically be applied to each database update, and the DBMS notify the user of the times and dates on which each data item was created and last accessed.

In Web applications, it may also be desirable to include similar creation/modification time/date information on frequently-changing Web pages (e.g., by including an HTML tag that indicates “last time updated” for a given page).

4.3.3 Least Privilege for Application Processes

Least privilege in applications is a function of how system privileges are granted to application processes by the operating system, as well as used and relinquished by those application processes. The application developer should ensure that the application as a whole, and each individual process in it, is not granted privileges that exceed the minimum privileges the application/process needs to perform its current operation. Moreover, the application and process must not retain any privilege after the operation it needed the privilege to perform is completed. For example, a process should not be granted write privileges if it requires only read access to complete an operation, even if the same process will later need write privileges. The write privileges should be granted only when they are actually needed and should be revoked as soon as they are no longer needed.

Another way to ensure least privilege is to develop the application so that very few if any of its component programs or processes ever need to be privileged or trusted.

If the application, particularly a classified application, defines user accounts, and any of those accounts are privileged, the application must provide role-based access control (RBAC) that can be used to enforce separation of duties and least privilege. See Section 4.3.4.

4.3.3.1 Separation of Duties

Assign only appropriate system roles to application processes, that is, roles that grant to each process only the privileges it needs to perform its functions. Separate system roles should be created for privileged processes and unprivileged processes, at a minimum.

4.3.3.2 Separation of Roles and Separation of Privileges

The roles assigned to application processes (and recognized by application access controls) should directly correspond to the various duties assigned to those application processes. Granting a particular privilege to only a very few application processes will make it much easier to determine whether those privileges are being used securely. As noted earlier, a process should always give up a privilege immediately after using it.

Implement commands that require different privileges as separate, small, simple executables that call each other, instead of a large, complex executable containing multiple commands. By doing so, each command will require fewer changes of privilege level during its use. Often, a series of multiple small, untrusted programs can accomplish the same task that a single large, complex, trusted program would otherwise perform. The advantage of an application made up of small, simple executables is that it will be easier to troubleshoot problems (both in general operation and in privilege handling), and much easier to accredit.

4.3.3.3 Minimizing Resources Available to Application Processes

Minimize the computer resources that are made available to each process (For example, use `ulimit()`, `getrlimit()`, `setrlimit()`, `getrusage()`, `sysconf()`, `quota()`,

`quotactl()`, and `quotaon()` in UNIX; there is also `pam_limits` for PAM processes). In this way you can limit the potential damage if a particular process fails or is compromised. This approach will help prevent denial of service attacks on the application. In Web servers, set up a separate process to handle each session and limit the amount of CPU time, and the like, that the session in each process can use. In this way, any attacker request that hogs memory or CPU will be prevented from interfering with any tasks other than in its own session. Although an attacker can establish many sessions, designing the sessions to be atomic and resource limited will make it more difficult for the attacker to achieve denial of service.

4.3.3.4 Separation of Domains (Compartmentalization)

Separation of roles and privileges is difficult to implement if there is inadequate separation of domains. Together, the two controls ensure that users and processes are able to perform only tasks that are absolutely required, and to perform them on data, in memory space, using only those application processes, and so forth to which they absolutely must be granted access to accomplish those tasks. Compartmentalizing users, processes, and data also helps contain any errors and violations.

Note that NT and most UNIX (and Linux) access controls cannot isolate intentionally cooperating programs. If the cooperation of malicious programs is a concern, the application should be implemented on a system such as Trusted Solaris, which implements mandatory access controls and limits covert channels.

4.3.3.5 Least Privilege in Web Applications

Programs and scripts that run as the Web server's "nobody" user should be modified to run under a specific username. "Nobody" is an anonymous account and should be disabled on the Web server.

Do not allow the program or script to create files in world-writable directories, such as the `/tmp` directory. See Section 4.4.1 for a detailed discussion of handling temporary files and directories.

Minimize the amount of data that can be accessed by the user. For example, in CGI scripts, place all data used by the CGI script outside of the document tree if there is no reason for users to see those data directly. Not providing an explicit link to the data will not prevent determined users from accessing the data.

4.3.3.5.1 Least Privilege and Java Sandboxes

Though Java applets are run in a sandbox by the browser, whether developing one's own applets, or using COTS applets, the developer should determine and implement a security policy that minimizes the privileges granted to the applet, that is, implementing least privilege for the applets.

In addition to least privilege, however, a sandbox mechanism can be a desirable way to limit the extent of possible impact that even non-Java applications may have on their operating environment. Appendix C includes some non-Java sandbox tools.

4.3.3.6 *Least Privilege in Database Applications*

If the Web application calls a database (e.g., via a SQL query interface), limit the rights of the application's database user role to only the specific set of stored procedures explicitly defined for that user. Implement as much functionality as possible within that set of stored procedures, so that if a hacker manages to force arbitrary strings into a SQL query, the resulting damage will be limited to that set of procedures only.

Also, if the application must direct a regular SQL query using client-supplied data, wrap the query in a wrapper that limits its activities (e.g., `sp_sqlexec`).

Most major RDBMSs provide packages, libraries and APIs that enable them to interface, as backend databases, with Web frontends. For example, Oracle provides Java and PL/SQL Web interfaces, as well as the fully integrated Oracle Web application server, a Web server through which clients can access both static HTML pages and dynamic content stored in Oracle databases.

4.3.4 **Role-Based Access Control**

RBAC is a nondiscretionary access control mechanism that allows and promotes central administration of an organizational security policy. The National Institute of Standards and Technology (NIST) is sponsoring a great deal of research in the area of RBAC, including RBAC for Web servers. See its site at <http://csrc.nist.gov/rbac/#servers>. NIST has also developed a proposed standard for RBAC, and one of the first CC protection profiles published specifies the evaluation criteria for RBAC.

RBAC needs to be implemented at a minimum for the privileged users and accounts of the Web application, and it must enforce separation of roles and least privilege (imperative for classified applications). If not supported by the operating system, the RBAC logic may be embedded into the application to increase the granularity of access control and enable enforcement of application-specific policies. Embedding RBAC into the application itself has the disadvantages of difficulty and expense to develop, lack of consistency in the definition of roles and enforcement of RBAC policy across applications that may touch the same data, and lack of scalability (i.e., to accommodate changes in the size and distribution of the user community).

One alternative to embedding RBAC logic into the application is to integrate an RBAC add-on into the underlying operating system. NIST has sponsored development of RBAC operating system add-ons for UNIX and Windows NT. Appendix C provides links to this and other add-on RBAC systems.

A second alternative is to implement access control middleware, such as common object request broker architecture (CORBA) access control middleware. This middleware provides a scalable, consistent external RBAC capability that can be used by multiple applications in the operating environment. Access control middleware is less difficult and expensive to implement than embedded RBAC, and it is easier to evolve and administer over time. However, the granularity of the policies supported is inferior to the granularity that can be custom defined in embedded RBAC; because the middleware is COTS, its RBAC policy-definition is fairly generic because of its need to support many different organizations'

policies. For most Web applications, however, the granularity and specificity of RBAC policies supportable by middleware will probably be sufficient.

4.3.5 Additional Web Content Access Control Measures

Some further access control measures may be implemented to protect Web content from being copied and misused (e.g., Web page defacement; plagiarism of Web content). They inhibit users from

- Copying HTML source code
- Cutting and pasting text content
- Screen capture.

Notice that we said “inhibit” rather than “prevent,” because it is probably impossible to absolutely prevent a determined plagiarist or hacker from printing the content of a Web page and scanning it with an optical character reader or retyping it. However, these techniques should help inhibit the less determined, less resourceful hackers and casual plagiarists from performing these actions.

4.3.5.1 Inhibit Copying of HTML Source Code

To inhibit copying of HTML source code, consider use of an HTML authoring tool or add-on that enables source code encryption, such as Authentica’s NetRecall and Andreas Wulf Software’s HTML Guard (see Appendix C).

Unfortunately, there is no way the Web server can prevent the browser from being able to display HTML source code in the first place. The user can choose to turn off source viewing in the browser, but this cannot be controlled by the server. If HTML source viewing is seen as a major problem, you may want to code the main elements of the Web site (navigation, header, etc.) in a Java applet, if possible, instead of using HTML. Unlike HTML source, Java applet source code, as well as CGI source code cannot be displayed using a browser’s View Source function.

4.3.5.2 Inhibit Cutting and Pasting of Text Content

Another countermeasure for inhibiting the cutting and pasting of Web content is to serve content as Postscript Document Format (PDF) files instead of HTML. Although it possible to cut and paste PDF files downloaded and displayed in an Acrobat Reader (versus via the Acrobat browser plug-in, which does prevent PDF cut and paste), the process is somewhat awkward and may inhibit casual plagiarism.

An even more effective approach is to scan any text documents that you do not want to be cut and paste into .GIF or .JPG files, and serve them as images rather than text. While these image files in their entirety can be copied and pasted, the text within them cannot be extracted electronically.

None of these technical countermeasures can prevent a user from printing a hard copy of a Web page (even one posted as an image file) and scanning it with an optical character reader. However, some

COTS products designed to control copyright-protected material do advertise the ability to prevent browsers from sending protected Web content to a printer. But even such products cannot prevent a determined user from simply rekeying the text contained in a Web page.

4.3.5.3 Inhibiting Screen Capture

To prevent screen capture by a browser, write a plug-in that wraps the system-level commands that implement the browser's screen capture function. This "wrapper" plug-in, when installed in the browser, will effectively disable those commands and thus, prevent the screen capture process from occurring.

4.3.6 Labeling and Marking of Output (Displayed and Printed)

If the Web application handles Privacy Act data, proprietary data, or classified data, labeling or marking of output by the application will be required. For labeling of Web pages, one approach is to use metatags to store the label information directly in the HTML source of the page, to be displayed as part of the page in the user's browser. If the Web page is printed from the browser, this label will be included in the printed output.

4.3.6.1 Platform for Internet Content Selection Labels

Another possibility is the use of a platform for Internet content selection (PICS)-compliant labels within HTML source code to implement sensitivity labels. Invented to add filterable adult content and similar warnings to Web pages, PICS can be used to define and apply other labels, including sensitivity labels.

PICS should be viewed as a convenience—a mechanism that can be used to define and apply informational sensitivity labels to Web pages. The labels would be displayed on those pages by Web servers that are able to distribute PICS labels along with Web documents. There are also proxy servers designed to filter Web content based on PICS rules. Such filtering can be used to prevent distribution of pages with a certain label to any users who are not authorized to see information with that label. However, PICS should not be seen as an alternative to true mandatory access control in applications that need it. See <http://www.w3.org/PICS/> for more information about PICS and how it can be implemented to add informational sensitivity labels to Web pages and distribute them based on those labels.

4.3.7 Encryption of Data at Rest to Augment Access Controls

If the underlying access controls are not considered trustworthy to protect sensitive data handled by the application, the developer may write the application to encrypt any data it saves to the underlying Web server file system or backend DBMS.

Encryption of data at rest to augment the access controls meets several security objectives that Web server or DBMS access controls alone cannot meet:

- Protection of data from the insider threat
- Data privacy

- Need-to-know separation in non-compartmented systems.

Appendix C lists several tools for encryption of data at rest.

4.3.8 Session Control

4.3.8.1 Session Management Schemes

Because HTTP is a stateless protocol, the application must include a session token scheme that enables all of a user's multiple requests to be associated with each other within a session. Each user request or transaction is authenticated by the user's session token.

After the user's browser is authenticated by the Web server, a session token is transmitted from the Web server to the browser embedded in a cookie, in a static or dynamically generated URL, or as a hidden field in an HTML page. If cookies, URLs, or HTML are used to convey session tokens, they should be transmitted using HTTPS (rather than HTTP) over an encrypted channel created using SSL/TLS. Encryption of the connection will protect the session tokens from capture and replay or tampering by an outside attacker. An attacker could use the captured token to insert himself into an existing session without authentication.

The session management scheme used in the Web application must enable the administrator to define the maximum number of simultaneous sessions to be allowed to each username, the session time-out (see Section 4.3.8.2), and other aspects of how the session should be established, maintained, and managed.

4.3.8.2 Session Time-Out

Authenticated session tokens on the Web server should be forced to expire, either after a fixed period of time, or after a maximum number of requests and transactions have been performed. Token time-out minimizes the possibility of a token being hijacked or brute-force attacked. Application session time-outs should be implemented in both directions: client to server and server to client.

Time-out of a token may mean that the valid user will have to reauthenticate to the application to receive another token. Or the server may transparently (without user reauthentication) do time-out and regenerate the user's session token. In either case, token time-out narrows the window during which an attacker can exploit a token to hijack a session. In addition, whether the current session token has expired or not, when a user tries to perform a sensitive transaction—such as sensitive data transfer, financial transaction—the server should require the user to reauthenticate or, at a minimum, should expire and reissue the user another session token immediately before allowing the sensitive transaction.

The session management capability of the application must also allow the administrator to set an inactivity time-out threshold, that is, the number of minutes without any user interaction with the server application, beyond which a session time-out should be instigated. If this inactivity time-out is not set, or if it is of longer duration than the session token time-out, the token expiration will do a time-out the session anyway. For this reason, you may want to write the inactivity time-out configuration process to

offer the administrator several options for inactivity duration before time-out, with the longest selectable duration being equal to the duration of the session token.

Finally, when the user logs out of the application (or a session inactivity time-out occurs), the application should overwrite any session cookies and other mechanisms used to store session tokens in the user's browser. That would prevent another person from sitting down at the user's workstation in the user's absence and modifying or reusing a stored session token.

4.4 IMPLEMENTING CONFIDENTIALITY

4.4.1 Application Support for Object Reuse

In addition to the object reuse capabilities of the underlying operating system and the backend DBMS (if any), the Web application itself should provide some basic capabilities for overwriting sensitive data in memory immediately after the data are used. That would minimize the possibility that they may be disclosed to unauthorized users. Such sensitive data include passwords, secret keys, session keys, private keys, or any other highly sensitive and secret data. Furthermore, applications should be scrupulously correct in deleting at the end of a session all temporary files they create during the session. Finally, application programs should not be allowed to generate core dumps when they fail.

4.4.1.1 Object Reuse in Java Applications

In Java, do not use the type *String* to store a password, because *String* is immutable—that is, they are not overwritten until garbage collection is invoked. Even after garbage collection, *String* may not be reused until sometime in the distant future. Instead of *String*, use *char[]* to store secret data in memory; this will ensure that the data will be immediately overwritten after use.

4.4.1.2 Avoiding Inadvertent Copies of Sensitive Data

Avoid writing application functions that create temporary files or file copies. If such a function cannot be avoided, ensure that the temporary files and copies are purged from the disk and from memory as soon as the processes that create them and use them are terminated. Ensure that the entire file is actually erased or overwritten (e.g., with zeroes), not just the pointer to the file.

4.4.1.3 Preventing Core Dumps of Sensitive Data

Programs should not be allowed to perform core dumps except during testing. Although core files can fill up a file system, they may also contain confidential information that can be discovered by an attacker who forces a core dump. For example, if a program fails, by default many operating systems create a core file that saves all of the program's memory. Depending on when the failure occurs, this file may include passwords or other sensitive data remnant in memory at the time of the failure. In some cases, an attacker can actually use the fact that a program dumps core to break into a system.

Instead of dumping core when a program fails, have the program log the appropriate problem and exit. In addition, limit the size of the core file to 0 (e.g., using *setrlimit* or *ulimit* in UNIX), to disable

creation of cores so that an attacker who halts the program will not be able to find any secret values in the core dump.

4.4.2 Confidentiality of Configuration Data and *include* Files

Place *include* files and configuration files outside of the Web documentation root in the Web server directory tree. This will prevent the Web server from serving these files as Web pages. For example, on the Apache Web server, add a *handler* or an *action* for *.inc* (*include*) files:

```
<Files *.inc> Order allow,deny Deny from all</Files>
```

Place the *include* files in a protected directory (e.g., *.htaccess*), and designate them as files that will not be served. Also, use a filter to deny access to the files. For example, on the Apache Web server use:

```
<Files ~ "\.phpincludes"> Order allow,deny Deny from  
all</Files>
```

If full regular expressions must match filenames, in Apache use the `FilesMatch` directive.

If the *include* file is a valid script file to be parsed by the server, make sure it is designed securely, and does not act on user-supplied parameters. In addition, change all file permissions to eliminate any world-readable permissions. Ideally, the permissions will be set so that only the uid/gid of the Web server can read the files.

Unfortunately, such permissions limitations can be circumvented by an attacker who is able to get the Web server to run his own scripts to access the files. One countermeasure to this problem is to run different copies of the Web server program: one for trusted users, and a second for untrusted users, each with appropriate permissions. This approach is difficult to administer, however. Yet if the perceived threat is great, the additional administrative overhead may be worth it.

4.4.3 Confidentiality of User Identities

Neither the Web server application nor the browser should transmit user identity information in the clear. User identities, and any other personal information should only and always be transmitted over SSL-encrypted channels. In addition, user identity and personal information stored by the Web server should be strongly protected by the server's access controls and perhaps also by stored-data encryption.

4.4.3.1 Do Not Hard Code User Credentials

Avoid the unfavorable practice of hard coding credentials in a Web page or in any other source file. Instead, store user credentials and other sensitive security information centrally, on an adequately protected server that is audited.

4.4.3.2 Pass Sensitive Data Using POST, not GET

Do not pass sensitive data as parameters in *GET* query strings. When sensitive data are to be passed to the server, do not send them as a parameter in the query string. For example, do not structure URLs as illustrated below:

```
http://www.site.gov/process_card.asp?cardnumber=1234  
567890123456
```

Because HTTP requests are logged in to the logs of the Web server as well as by whatever proxies exist between the browser and the server (including transparent proxies), even if the request in the HTTP string is sent via SSL, the request will be decrypted and logged as cleartext. Or it may be Base64 encoded but not encrypted. Here is an example:

```
2000-03-22 00:26:40 - W3SVC1 GET /process_card.asp  
cardnumber=1234567890123456 200 0 623 360 570 80  
HTTP/1.1  
Mozilla/4.0+(compatible;+MSIE+5.01;+Windows+NT) - -
```

In addition, the entire URL may be stored by the browser in the browser's history file, potentially exposing the sensitive information to anyone who can access the client system. Instead of *GET*, use *POST*, as follows:

1. Use the HTTP body to pass the sensitive information. The HTTP body is not logged, and thus the log will not contain sensitive parameters.
2. Use SSL (with 128-bit minimum key length) to encrypt the information in transit.
3. If the data are truly sensitive, make sure they are encrypted when stored, both on the sending (client) and receiving (server) sides.

4.4.3.3 Exclude Confidential Data from Redirects

HTTP includes several redirect response types, telling the user that a Web page that no longer exists at the URL submitted by the user has moved to a different location and in many cases automatically redirecting the user's session/connection to that new location. Attackers can hijack valid HTTP redirects or inject spurious redirects to reroute the user's session or connection to a Web page that triggers malicious code that causes illicit actions to be taken on behalf of legitimate users who think they have been redirected to a valid Web page.

A specific security problem arises when a Web page that causes a redirect can be accessed only after the user is authenticated and authorized. If basic authentication or HTML forms-based authentication with cookies is used, the user's browser will store that password or authentication cookie and present it each time the accessed site demands reauthentication of the user and session.

If an attacker manages to maliciously redirect the user to a Web page under the attacker's control, that page may be programmed to look like a legitimate page, and to perform the action the user expects. So the user's browser will provide that page with the user's credentials, which it has stored. In this way, the attacker is able to collect the credentials of unsuspecting users.

The potential for this type of attack is yet another reason that DoD Web applications should not use Web server basic authentication nor HTML-forms authentication with credentials subsequently stored in a browser cookies. Authentication based on X.509 certificates transmitted via SSL is not vulnerable to this type of attack.

4.4.4 Validate URL Pathname Extensions

A URL, at its simplest, is the pathname to the file system directory location of a Web page on the Web server. However, in some Web applications, this pathname is appended with other information, such as a query part containing variables that instruct the Web server to handle the user's access request in a certain way. Here is an example:

```
http://www.org.gov.mil/PKIsystems/platform/tech-features.xml?page=9
```

In that URL, the user has requested the file *tech-features.xml*, which is stored in the directory */PKIsystems/platform/* on the Web server host *org.gov.mil*. By appending the URL extension “*?page=9*”, the URL requests the server to return not just the whole document *tech-features.xml*, but the specific page (9) within the document.

URL extensions can be used to instruct the Web server to do many actions. For example, consider the following URL:

```
https://www.CA.gov.mil/cgi-bin/check-rev.cgi?02a56c
```

The problem with URL extensions arises when a hacker intercepts a URL in transit from browser to server and modifies the URL extension to request the server to perform an action that is unexpected or destructive. For this reason, the Web server application must be able to recognize questionable URL extensions sent to it by browsers, to validate the URL to ensure it does not contain such an extension, and to truncate a URL after the end of the file pathname to remove any extension that does seem suspicious.

A specific threat related to URL extensions is that of denial of service. On certain Microsoft IIS servers, an attacker can increase the Web server's CPU use by requesting a complex URL that includes intentionally malformed extension information, such as a large number of dots and slashes. IIS versions 4.0 and 5.0 are vulnerable to this form of denial of service attack. Microsoft has released a patch for this vulnerability.

4.4.5 Limit Data Returned to Users

Limit the data returned to the user to what is specifically requested by the user. Avoid giving more than the bare minimum of information to untrusted users about application processing status. Report only that a transaction or process has succeeded or failed, possibly with a brief, generic explanation of the failure.

4.4.6 Do Not Trust Browsers to Store Sensitive Data

Never treat a browser as trustworthy rely on a browser for storing important or sensitive data. Such data should be stored on the server and protected appropriately. If the browser frequently needs important nonsensitive data, it should store a copy obtained from the server in cache at the beginning of a session. It should also be configured to purge cache at the end of a session (when the browser is shut down). Sensitive data should not be stored on browser platforms at all unless there is a mechanism for storing the data in encrypted form, with trustworthy management of cryptokeys.

The message will be repeated several times in this document: Never trust a browser in any security-related or sensitive transaction.

4.4.7 Application Integration with Data Encryption Mechanisms

For some application functions, performance requirements may entail the use of an encryption implementation that is faster and more efficient than the symmetric encryption capability embedded within the Web application's DoD PKI implementation. Appendix C lists some encryption technologies approved by the NSA that may be integrated into applications.

When evaluating encryption products for use in the Web application, consider whether the product can be easily integrated into, or called from, the application. Consider several questions. Does the product include its own API or call-level interface (e.g., in a software library), or will this interface have to be custom-implemented by the developer? Also, does the encryption product—including its API/interface to the application—strictly protect the data to be encrypted from disclosure while those data are in cleartext form? Does the product adequately protect the encryption keys and key management material it uses from interception and tampering? Does it perform key management and certificate management and handling correctly, such as correctly responding to and acting on key revocation lists, certificate revocation lists, and on-line certificate status protocol (OCSP) transactions.

If the cryptographic material used by the encryption product is stored and retrieved from a directory (as is the case with the DoD PKI), the underlying file system access controls, as well as the directory's own access protections, must be configured correctly to protect that cryptographic material from compromise. In the case of the LDAP directories used in the DoD PKI (and in most other PKIs), it has been reasonably argued that neither the directories themselves, nor LDAP protocol, is truly secure. It is crucial that your Web application makes sure that the channel over which LDAP is used to transmit cryptographic materials (such as cryptokeys, certificates, key revocation lists, certificate revocation lists, OCSP transactions, etc.) are encrypted via SSL and in some cases also a VPN (such as IPsec) at the transport or IP layer.

Appendix C provides more information on the security criteria that you should use when evaluating third-party encryption and other third-party security technologies for inclusion in Web applications.

NOTE: The next version of this document will include a list of standard and preferred APIs and methods to be used in DoD applications.

4.4.7.1 Encryption Before Transmission

Web-based applications should encrypt all communication with a user that includes sensitive information by using the HTTPS protocol over an established SSL connection, rather than HTTP over a nonencrypted connection.

Nor should you use HTTP *GET* requests to submit forms that contain sensitive data, even when transmitting over an SSL-encrypted connection. *GET* causes the data in the HTTP request to be encoded in the Request-URI (that is, uniform resource identifier), which many Web servers, proxies, and user agents log in cleartext—a target for hackers seeking sensitive user data. Instead of *GET* requests, use HTTP *POST*, which is intended specifically for submitting HTML forms without encoding the contents in the Request-URI.

4.4.7.2 Encryption of Data at Rest

The application should encrypt any sensitive data that it stores in a database, file system directory, or backup that does not provide sufficiently strong access controls to protect those data from possible compromise. This is particularly important for backup disks that may be stored or transported without sufficient physical protection.

The cryptokeys for decrypting the stored data should not be stored on the same physical system or medium that contains the encrypted data.

For information on use of encryption of data at rest to augment inadequate access controls, refer to Section 4.3.7.

4.5 IMPLEMENTING DATA AND CODE INTEGRITY MECHANISMS

Integrity of data and mobile code transmitted or stored by DoD Web applications may be assured using electronic integrity mechanisms such as hash or digital signature. These electronic integrity mechanisms must be implemented using DoD-mandated (or recommended) technologies. In the case of hash, the DoD and the U.S. Federal Government in general have both mandated the use of the SHA-1 hash algorithm (FIPS PUB 180-1), and not MD5, whenever secure hash is required.

The DoD PKI supports RSA as its digital signature algorithm and provides the option of using the FIPS Digital Signature Standard's DSA algorithm. Because both algorithms are to be supported by DoD PKI, the digital signature validation software implemented in DoD Web applications should be able to validate digital signatures generated using either RSA or DSA.

The technical implementation of a digital signature mechanism and a signature validation mechanism for protection and verification of data integrity will be the same as the implementation of digital signature for nonrepudiation. See Section 4.7 for details.

4.5.1 Implementing Hash

To protect the integrity of data transmitted between server and client, the two entities need to perform an integrity check in addition to encryption. That is because encryption does not prevent an attacker from randomly changing the encrypted data stream, which will transform the data it contains into a collection of nonsensical characters.

The standard cryptographic integrity mechanism supported by the DoD PKI is the hash, a calculation based on characteristics of the data to be protected, It is appended to the transmitted data so that the recipient can compare the transmitted calculation appended on the data sent with the new calculation he performs on the same data, using the same calculation technique. If the two calculations—the one sent, and the new recalculation—result in the same hash value, the data are proven not to have been changed during the transmission.

It is not adequate to use the hash function directly to calculate the hash value to be appended to the data. That could expose the data to an extension attack in which an attacker can use the hash value to derive the calculation used, add his own data to the file and then compute and apply a new hash to the data. The recipient would not be able to detect the change of hash.

Instead of hash alone, a hashed message authentication codes (MAC) better ensures the integrity of data transmitted between two entities. A MAC based on a cryptographic hash function that uses a secret key available to both entities is called an HMAC (Hash-MAC). RFC2104 defines the cryptographic formula for HMAC as

$$H(k \text{ xor opad}, H(k \text{ xor ipad}, data)).$$

where H is the hash function (in DoD PKI, this will be SHA-1), k is the shared secret key, and $data$ is the data to be protected. The integrity mechanism for DoD applications is referred to HMAC-SHA-1. HMAC is at least as strong as the hash function it uses.

4.5.1.1 Hashing Data to Prevent Tampering

To prevent users from tampering with transmitted data, we are providing an example of hashing an HTML form, with the hash used to prevent tampering with, as well as hijacking and replay of, that form.

NOTE: The process for calculating and applying the HMAC to data is the same, regardless of the type of data to be hashed.

1. Implement a message authentication check (MAC) on the HTML form, using a SHA-1 hash.
2. Calculate the initial hash using a secret hash key available to both the browser and the server. This hash key should be of a length appropriate for the sensitivity and mission criticality of the

data being hashed (refer to Information Assurance Technical Framework [IATF] Version 3.0, Section 4.5, for specific key lengths to be used when hashing data of different mission criticalities and sensitivities). Also, the hash key should contain upper and lower case alphanumeric characters that do not spell out a word in any human language.

3. Determine which form fields should be hashed to make them tamperproof.
4. Add hidden fields that contain consistency-checking information to the form. Such information may include
 - Expiration time for the form
 - Name of the script that is supposed to process the form
 - User's IP address.
5. Concatenate the secret key, the contents of the tamperproof fields, and the consistency-checking information in the hidden fields into one long string. That concatenation comprises the MAC for this form.
6. Compute the hash of this concatenated data.
7. Convert the computed hash into a printable hexadecimal string.
8. Include within the form a new hidden field containing this hash.

When the form is returned to the server-side script by the user, the server application should do the following:

1. Recover from the form: the hash, the tamperproof fields, and the consistency-checking fields, making sure that all expected fields are present.
2. Verify that the consistency-checking fields match their expected values.
3. Recompute the hash using the same formula as before and compare the recomputed hash to the hash retrieved from the form.
4. Recognize that if the two hashes match, it is highly unlikely that the form has been tampered with or replayed.

4.5.2 Integration of Digital Signature Mechanisms

4.5.2.1 Interface from Application to Digital Signature Mechanism

Depending on the language in which the application is being written, the language itself may provide a predefined API or call interface to the digital signature mechanism. For example, in Java, the

java.security package includes the public abstract classes *SignatureSpi*—the service provider interface (SPI) to a digital signature algorithm—and *Signature*—which is an extension to *SignatureSpi*. See <http://klomp.org/mark/classpath/gjdoc/java.security.SignatureSpi.html> and <http://klomp.org/mark/classpath/gjdoc/java.security.Signature.html>.

Check the constructs of the programming languages in which the application is being written to determine whether system calls to digital signature are provided. If they are not, the digital signature mechanism itself should provide an API as part of the software developer kit (SDK) supplied by the vendor.

4.5.2.2 Digital Signature and Validation Capabilities for Browsers

Digital signature and signature validation in browsers is supported by various plug-ins, some of which are specific to the type of content to be digitally signed or validated. The digital signature capability in the browser will be used mainly by users to digitally sign HTML forms content before transmitting those forms to the server.

Browser validation of digital signatures will be used mainly to validate signatures on mobile code. For example, Netscape Communicator browsers offer an optional HTML form signing capability. The Java Plugin 1.3 enables user verification of RSA digital signatures applied to Java applets downloaded to the browser, so that the user can validate the signature before deciding whether to allow or disallow execution of the applet.

A list of HTML digital signature and code signature validation browser plug-ins appears in Appendix C.

If the browser is also used as the e-mail client, as with Netscape Mail, the client mail application should provide security functions that support digital signature of e-mail messages. It should use DoD PKI certificates and validation of certificates and digital signatures on received messages.

4.5.2.3 Digital Signature Validation by Server Applications

The digital signature validation capability in the server application must correspond with the digital signatures provided by the browser. For example, if signed HTML forms from Netscape browsers are expected, the server application must have access to the Netscape signature verification tool on the Web server to process those signed transactions.

Another alternative is have digital signature validation performed on behalf of the application by a separate signature validation system, such as KyberPass's Validation TrustPlatform (see Appendix C)

4.5.2.4 Protection of Cryptographic Material Used for Digital Signature

The cryptographic material (certificates or keys) used by applications for digital signature should be protected to the same extent and in the same way as are the cryptographic materials used for authentication, encryption, and hash.

4.5.2.5 Preventing Web Page Defacement

To avoid the problem of Web site defacement, the server should never accept or post (publish) HTML files or other Web content until that content has been proven to have originated from a known, trusted source authorized to publish content on the Web server. Neither the originator's IP address nor a Web form referrer header constitutes a trustworthy proof of origin, for both IP addresses and referrer headers can be tampered with.

To ensure that Web content to be published comes from a trustworthy source, require all valid Web content authors to digitally sign their files. The Web server should validate the digital signature on each file when it is uploaded to the server and reject any files that have not been digitally signed by an authorized Web author. In this way, the digital signature acts as an integrity mechanism to prevent serving of defaced Web pages that hackers may have stored on the server.

A high-end system that is specifically designed to prevent serving of fraudulent Web content is Gilian Technologies' ExitControl, and particularly its G-Server ExitControl appliance. Instead of relying on users to validate digital signatures on Web content, the G-Server validates those signatures to verify that Web pages have not been tampered. It does so before serving those pages to the user's browser. See Appendix C for more information.

4.5.2.5.1 Digital Signature of Web Content

There may be instances in which Web authors wish to digitally sign Web content, to prove its authenticity or provide nonrepudiation of its source, as well as to provide an integrity mechanism to prevent serving of defaced Web pages. HTML, XML, PDF files, and even backend database entries may need to be digitally signed before that content is placed on the Web server to be made accessible to users via the Web application. This digital signature capability will be of interest to Web application developers as well, if they are also responsible for the integrity of the Web content presented by their applications.

A specific discussion of XML digital signature based on the new XML digital signature standard appears in the discussion of XML security in Appendix D. Some products for enabling digital signature of Web content are listed in Appendix C.

4.5.3 Code Integrity

All application code running on the Web server should interact with the underlying middleware or operating system layer cryptographic system that performs digital signatures and encryption. To implement digital signatures on application code, follow these steps:

1. Have the application create a hash-object by calling a function that initiates the hashing algorithm for a stream of data. This function will return a handle for the application to use in subsequent calls.

2. Have the application add data to the hash-object, and then call a digital signature function in middleware or the operating system to sign the hash value.
3. After the middleware or operating system layer accepts the call and signs the value, then encrypt it using the signature private key.

In addition to using hashing and signing program code, the previously mentioned security mechanism known as a sandbox can be implemented. It is a standard part of Java and Perl, but it may be implemented for other languages using third-party sandboxing mechanisms. A sandbox is used to assign a level of permission (or privilege) at run time to the application executable that will run in that sandbox. The sandbox permission granted to an executable will depend on what operations the application is expected to perform during its normal, correct operation. If the application attempts to run in an unexpected way, the sandbox will generate an exception that will restrict the code and prevent it from performing the illegal operation or accessing any resources outside the sandbox.

4.5.3.1 Digital Signature of Mobile Code

4.5.3.1.1 Java Applets

Digital signatures on Java applets are verified by the Java Plugin 1.3 in the browser (see <http://java.sun.com/products/plugin/1.3/docs/nsobjsigning.html>), which can verify RSA signatures on Java applets via the plug-in's Cryptographic Service Provider (CSP). The CSP should be configured to use the *SHA1withRSA* digital signature option. This algorithm will be automatically registered with the Java Cryptographic Architecture framework as part of the static initializer of the *PluginClassLoader*.

If the applet is signed, and the permissions granted to it do not include the special *usePolicy* permission, the *PluginClassLoader* will extract the signers (and their supporting certificate chains) from the applets source code and attempt to verify them.

If the Java plug-in can verify the certificate chain all the way up to its root CA certificate, it will also verify that the root CA certificate is contained in the database of trusted root CA certificates. If so, the plug-in will display the certificate chain of the authenticated signer and ask the user whether or not to grant *AllPermission* to code signed by that principal. Java code that is assigned the *AllPermission* permission is treated in the same way as system code is, meaning it has all the privileges that system code has. The user can then choose whether or not to grant *AllPermission* to code signed by that principal and whether such permission should be granted to any code signed by that principal for all subsequent sessions or for the current session only.

4.6 ACCOUNTABILITY OF APPLICATION USERS

In addition to auditing at the operating system and DBMS levels, the Web application needs to log security-relevant events and store that log data securely to prevent unauthorized tampering or disclosure of the log data. This secure storage may be implemented by channeling the log data via a secure

connection to an external audit system, such as a central audit collection server, audit middleware, or the operating system audit trail.

Read-only media for storing of audit data can help ensure nontamperability of the audit trail, but the only way to ensure nonbypassability of audit is by implementing audit data collection on a trusted system. Otherwise, accountability can be subverted by tampering with the underlying system, below the application layer at which auditing takes place. For example, auditing can be subverted by embedding code that captures audit records (data collecting trapdoors) or modifies or deletes the content audit log records.

4.6.1 Application Integration with System Audit Log or Audit Middleware

Write the application to log its own security-relevant events and errors. Also, structure this application logging capability so that it can reformat and pipe its log records into the underlying operating system audit trail, or into a middleware audit collection system.

For example, Apache's Log4j framework provides an extensible event logging service to applications running on the Apache Web server. Logging and audit entries captured by Log4j may be routed to the local operating system audit trail, or to audit middleware, using Log4j's pluggable logging modules. Appendix C lists some third-party audit middleware.

4.6.1.1 Minimum Requirements for Application-Level Audit

Whatever approach is used—integration with the underlying operating system's audit or with audit middleware—auditing of application security events must comply with all DoD-defined requirements for auditing in systems of the particular sensitivity level and classification and mission criticality at which the application will operate. These requirements include such capabilities as

- Support for administrator configuration of audit parameters, including events to be audited, information to be captured about those events, and setting of notifications and alarms on an event-by-event basis
- Support for administrator configuration of fill thresholds and disposition of nearly full and full audit records
- Tools for viewing of audit records, audit reduction, and capturing and printing of audit reports
- Support for administrator configuration of application behavior in case of audit failure
- API or other hook between the audit mechanism and any intrusion detection or other security monitoring system.

See *Recommended Standard Application Security Requirements* Table 4-6 for the required operational characteristics of application-level audit facilities. These requirements should be used as criteria when evaluating third-party audit products.

4.6.1.2 Application Events to Be Audited

See *Recommended Standard Application Security Requirements*, Table 4-6, for the specific types of security-relevant application events that should be audited, as well as what information should be captured about those events.

4.6.1.3 Application Logging if the Underlying Audit System Becomes Unavailable

The application's logging capability must also be able to continue operating—storing the log data locally, with access control protections (and possibly in encrypted form)—in case of a failure of the external audit system. It may be desirable in some applications to have application log data both stored locally (in a secure way) and channeled to an external audit facility.

In addition, the application's logging mechanism must allow the administrator to configure a maximum fill threshold. That pertains to the maximum number of log events, or number of bytes of data, that can be stored by the application's log before it generates an alarm, triggers an orderly shutdown or takes other actions. See *Recommended Standard Application Security Requirements* Table 4-6 for the required operational characteristics of application-level audit facilities and their response to administrator-configured fill thresholds.

4.6.1.4 Protection of Application Log Data Before It Reaches External Audit System

If an external audit system is used to store application-level log data, the interface between the application's own event log and the external audit system must be secure, either by using an encrypted connection (channel) between the two, or by encrypting the log data before transferring them via an unencrypted channel (socket connection) or API. In the case of a central audit system or audit middleware, the secure interface will ideally be provided as a standard feature of that system and middleware. If it is not, the application will have to provide it in a way that can be accommodated by the external system and middleware, and that does not compromise the security of the application's log data in transit to that external system and middleware.

4.6.2 Application Security Violation Notifications

4.6.2.1 Application Integration with Intrusion/Violation Detection

We strongly recommend using a Web application server that is able to detect violations of security policy, such as access control or authorization violations, and to raise a security exception if such a violation is detected (with the ability to generate an alarm to the administrator if such an exception is raised, as well as auditing of all such exceptions). Java J2EE, for example, includes a security manager that can be used by the application. The security manager performs all policy checking on security-relevant events, detects any violations, and raises security exceptions accordingly.

4.7 NONREPUDIATION BY APPLICATION USERS

Digital signature is the mechanism that should be used in Web applications to ensure nonrepudiation by users. Digital signature mechanisms will be implemented the same way whether they are used for integrity or nonrepudiation or both (only the certificate used will vary). Refer to Section 4.5.2 for information on implementing digital signature in Web applications.

5.0 MAKING APPLICATIONS RESISTANT TO COMPROMISE AND DENIAL OF SERVICE

5.1 AVOIDING BUFFER OVERFLOW

Buffer overflows result from programming errors and testing failures. They are common to all operating systems, although not to all programming languages. More than half of the security advisories from CERT (<http://www.cert.org>), including the August 2001 Code Red, originate from buffer overflow vulnerabilities.

To cause a buffer overflow, an attacker sends a string of data that is longer than the fixed size of the memory buffer allocated to receive the data string. As a result, the data string overflows the buffer, filling up other areas of memory that have been allocated for other purposes. The most common result of buffer overflow is a denial of service.

In more sophisticated buffer overflow attacks, the attacker uses the opportunity created by the buffer overflow to replace the saved return pointer on the stack containing the buffer with another pointer value. The result is that instead of pointing to the original program that sent the call containing the overlong data string, the stack now points to (and causes to execute) a malicious program created by the attacker. This malicious program will run with the same privileges as do the original program, and it may be able to add and delete data or to run, modify, or replace programs the attacker would otherwise be unable to access. It is difficult to create, but if successful, this sophisticated attack can be easily distributed as a reusable script to other hackers.

Buffer overflows are almost exclusively limited to C and C++ applications, due to the lack of input validation in those languages. Using another programming language is an way to avoid buffer overflows. Perl, for example, automatically resizes memory arrays and is thus immune to buffer overflows. Java includes automatic bounds checking for buffers, and it does not allow pointer manipulation. Python is another language that prevents buffer overflows, as are Ada and Pascal.

Even if you do not write a single line of code in C or C++ libraries, it is extremely likely that at least some of the software libraries, APIs, and third-party you use will be written in these languages. Thus, simply not using these languages is not a 100% guarantee against buffer overflow. However, it will go a long way toward minimizing the likelihood of buffer overflows. Input validation must do the rest.

If you do write in C or C++, the way to avoid buffer is to verify that the length of any input data string does not exceed the defined bounds (length constraints) for the field into which the string was input. If possible, write the application to truncate input strings that exceed defined bounds; otherwise reject and discard them. Under no circumstances allow overlong strings to bleed into other areas. This is exactly what causes buffer overflows.

Here are some general guidelines for preventing buffer overflow in any language that does not do input validation:

- Always program the results buffer to be larger than the source buffer.
- Do not use any function that does not check array boundaries.
- Always check bounds on the length of data before they are copied into program buffers.
- Check bounds on all array and buffer accesses.
- Make sure that excessively long data elements are not passed into other libraries, because you cannot trust other programmers' code not to induce internal buffer overflows even if your own code avoids them.
- Be aware of problematic functions and system calls. Use them safely, (in ways that will not subject them to buffer overflows, or use safer alternatives.

If you do plan to write some or all of your application functionality in C or C++, please read, in Appendix D, the specific discussion of C/C++ buffer overflow and its prevention.

5.2 AVOIDING CROSS-SITE SCRIPTING

If data from an untrusted user will be forwarded by the application to a second user, make sure the data does not contain malicious code and does not point to malicious code that could be executed by the second user's browser. Also make sure that data originating on the server and output to users have not been corrupted with embedded malicious code or pointers to malicious scripts on other sites. Web applications written in HTML or XML are particularly vulnerable to this kind of attack, which goes by names such as "cross-site scripting," "malicious HTML tags," and "malicious content." Cross-site scripting can be used on a hostile server to link users to your server by providing those users with a link that contains a corrupted form. It can also be used to fool your own users into thinking that a hostile script originated from your server and thus can be trusted.

Web applications that permit users to include HTML in their input that may later be posted to other users should include controls that protect input from interception and tampering by hackers. Examples of these Web applications are e-mail clients that allow HTML-formatted email messages, bulletin boards, and guest book applications. Hackers might use the HTML tags to insert scripts, Java references to hostile applets, 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 unsuspecting user's system. Damage includes

- Exposure of SSL-encrypted connections
- Access to restricted Web sites via the attacked browser
- Violation of domain security policies

- Rendering Web pages unreadable or difficult to use (defacement by adding annoying banners and offensive material)
- Violations of privacy (e.g., by inserting a Web routine that monitors exactly who accesses a certain page; embedding malicious *FORM* tags in data input forms, then modifying the form to trick the user into revealing sensitive information)
- Causing denial-of-service attacks (e.g., by continuously creating new browser windows)
- Targeting specific vulnerabilities in scripting languages
- Causing buffer overflows

Applications should not accept any input, including form data, without first taking these actions:

- Validating data when received to ensure that only safe data are accepted and that all other data are rejected or
- Filtering data when received, to remove potentially dangerous characters from the text, and accepting the rest or
- Encoding potentially dangerous characters to prevent them from causing trouble; encoding may be performed when the data are received (if the application will not be doing any processing of the data), or just before they are forwarded to another user (after the application have processed the data).

Finally, if an e-mail client is used in the application (client or server), use a text-only e-mail client that does not allow embedded HTML in message bodies.

Be aware that buffer overflow (e.g., `gets ()` problems, stack overflows) is an equal opportunity problem for operating systems. Buffer overflow is as likely to happen on Windows and MacOS as on UNIX and Linux.

5.3 INPUT VALIDATION

The countermeasure that will go the longest way toward achieving the resistance of Web applications to compromise and denial of service is validation of data input to the Web server application by users or external processes.

The most widely reported Web application vulnerabilities—buffer overflow, cross-site scripting, null byte attacks, SQL injection attacks, and HTTP header manipulation—are all caused by Web server applications that accept invalid data from Web clients. All can be prevented by thorough, correct input validation of data by the server.

All arguments and other input data strings submitted by users, external processes, and in some cases by internal untrusted processes should be carefully reviewed by the Web server application to verify that the data conform to the formatting and content characteristics defined as allowable for those data. The server application should never trust any input from clients and little if any from other servers, and it should validate all values that originate externally to the application program itself, including arguments, environment variables, and parameters. The server should not trust any external entity that has not proved its trustworthiness. Browsers are vulnerable and unable to prove their trustworthiness. Other servers' trustworthiness must be proven on a server-by-server basis.

The Web server application must also distrust validations done by the browser. At best, browser validation of data should be seen as a first-line filter that might prevent some invalid data from being sent to the server. Browser validation cannot be trusted not because the browser's validation routines are faulty, but because the browser itself, and the data stream sent by it, are both easily compromised. Client-side validation can be bypassed or interfered with by a malicious user or hacker (unless the client application runs on a trusted operating system with access controls that protect the client executable and data from tampering). Thus, even if the browser gives the user input a clean bill of health, there remains a real risk that the data will be tampered with between the time they pass the browser's validation and they reach the server. For this reason, input validation should be done by a trusted process in the Web server application, even if the data have already been validated by the browser.

If the system in which the application will be deployed provides a common set security services to all applications, there may already exist a common component whose function is to filter all input and output. If so, write the application to invoke this common filtering component, instead of embedding a custom-developed filtering function into the application. If the filtering component does not perform all of the input validation checks you feel are necessary, you may need to augment it with your own input validation routines in the application itself. Similarly, if there is no common input validation service available to the application, you will have to develop your own input validation routines.

NOTE: For additional detailed information and examples on input validation, refer to the chapter on Input Validation in Secure Programming for Linux and UNIX HOWTO (see Appendix C).

5.3.1 Designing Applications to Make Input Validation Easier

It may be possible to implement some input validations using third-party validation routines. Appendix C lists some available third-party input validation software. In most cases, however, you will need to write your own scripts and routines for validating user input.

Following the guidelines below when writing the application will make it easier to input validation checking in your application.

5.3.1.1 Clearly Define Acceptable Input Characteristics

Unambiguously define the acceptable type, length, and format of every data string expected as input to the application, for example, of every field in every HTML form. Also unambiguously specify all valid values (or ranges or values) for every parameter expected as input to the application. These definitions will serve as the rules to be enforced by input validation checking, whether it is performed by an external validation service invoked by the application or by validation routines within the application itself.

5.3.1.2 Use Only Functions That Perform Bounds Checking

Do not use any function in the application that copies input data without first checking buffer lengths.

5.3.1.3 Pass Arguments in Environment Parameters

Do not allow the application to trust operating system environment variables. Instead, pass every argument to the application in an environment parameter.

5.3.1.4 Suspend Processing Until Input Is Validated

Write the application to suspend processing of any transaction in which input (including remote IP addresses and port numbers) is received over the network until that input has been validated by or on behalf of the application.

5.3.1.5 Do Not Invoke Untrusted Programs from Trusted Programs

Do not write trusted programs in the application to invoke untrusted programs. Write the trusted programs to verify the trustworthiness of all programs they do invoke, by requiring the invoked program code to be digitally signed or hashed. Write a routine that invokes a PKI function to validate the signature and hash, and only allow the trusted application to invoke the program if its signature and hash can be validated.

5.3.1.6 Use Only Independently Certified Third-Party Components

Write all trusted programs to distrust third-party—COTS or public domain (open source, shareware, freeware)—software components whose trustworthiness has not been independently verified through NIAP certification, NSA Trusted Product Evaluation Program (TPEP) evaluation, or other means. Independent verification of third-party components is required regardless of whether a third-party components' code has been signed and hashed. See Appendix C for more information on independent verifications and security criteria to use when evaluating third-party components.

5.3.1.7 Validate Data Before Copying to a Database

If the application is a frontend to a backend database, the application should validate all user input before copying the data into the database. Do not rely on the database to validate the user input. Refer to the discussion of SQL injection in Section 5.4.1.2.

5.3.1.8 Write Scripts to Check All Arguments

Write not only software routines, but also all CGI and shell scripts, to check all of their arguments.

5.3.1.9 Protect Cookies at Rest and in Transit

Although cookies are the preferred method to maintain state when using the stateless HTTP protocol and to store user preferences and other data, cookies should not be used to store sensitive data such as session tokens and arrays used to make authorization decisions. Both persistent and nonpersistent cookies, whether secure or insecure, can be modified by a malicious user who gains access to the browser, then transmitted to the server in a URL request. Even nonpersistent cookies, though less vulnerable to tampering, can be modified using a tool such as *Winhex*.

SSL/TLS protects cookies in transit, but it does not protect cookies stored in the browser. Instead of storing user properties in a cookie in the browser, store them in the user's session token in server-side cache, directory, or database. When the server application needs to check a user property, it can then reference the username in its own session table, which will point to the user's session token (stored in the server cache and database) containing the needed data and variables.

To prevent acceptance of tampered cookies, encrypt each cookie with a symmetric encryption algorithm or hash the cookie, and have the application compare hashes when the cookie is returned.

5.3.1.10 Do Not Use Hidden Fields for User Input

When a user makes a selection in an HTML form, by indicating a selection in a pull-down menu or checking a box, or by typing in text, the selection is usually stored as an HTML form field value and sent to the application in an HTTP GET or POST request. HTML can also store field values as hidden fields. Hidden fields are not echoed to the screen by the browser but are collected as parameters and submitted in the background when the form is submitted to the server. Hidden HTML form fields are a convenient way for developers to store data in the browser and to convey data between pages in wizard-type applications. However, the same vulnerabilities that apply to other HTML form fields also apply to hidden fields.

Form fields, whether selected, typed in, or hidden, can all be manipulated by the user to submit whatever values he or she chooses by saving the HTML form page, using the browser's view source option, saving or cutting and pasting the source into a text editor file, and then modifying it and reloading the modified page into the browser.

Consider an application that uses a simple form to enable user log-in. In an attempt to prevent possible buffer overflows caused by users entering overlong usernames and passwords, the developer may set the form field value `maxlength=x` for the username and password fields, (where x is the integer defining the maximum number of characters allowed in the field. This approach to preventing buffer overflow is inherently flawed, however, because the hacker can simply save the HTML source of the

page, remove the *maxlength* tag, reload the page in his browser, and then enter an extremely long value in the field to generate a buffer overflow. Other HTML form fields that hackers often target are

- Disabled: often removed by the hacker to enable and otherwise disabled function or value
- Read only: often removed to enable the hacker to write to an otherwise protected field
- Value: often changed to a different value than the one set by the developer

Now imagine that the application shown, in which the HTML form is used to enable user log-in, includes a hidden HTML tag behind the *login* form field, to associate additional parameters that will instruct the server application in how to interpret the user-supplied data in that form, such as this:

```
<input name="adminaccess" type="hidden" value="N">
```

This HTML tag indicates that the log-in data provided by the user should cause him to be logged in as a normal user, not as the administrator (i.e., he should not be given *adminaccess*). By tampering with the hidden field, changing the value *N* to *Y*, the hacker can trick the application into logging him in as the administrator.

To prevent HTML form field manipulation, instead of using hidden form fields to embed various parameters about the normal form field data, as with cookies, embed these parameters in the user's session token and store that token in a server-side cache, directory, or database. When the application needs to check the parameters associated with that user's data entries in the form, it will reference the user's username in its session table, which will point to the session token, containing the user's data and variables, in the server cache and database.

5.3.2 Rejection and Sanitization of Bad Input

Rejection of bad input data presumes that the application can recognize specific erroneous and malicious inputs. In fact, it is very difficult for an application to maintain a complete, up-to-date database of malicious code signatures. If possible, the application should be implemented to invoke an external virus scanner and malicious code detector as part of its input validation process, though even the signature databases of the best scanners will not be up to date at all times.

Attempting to make bad input data harmless through sanitization should be seen as the second, not the first, line of defense. If input appears suspicious but cannot be authoritatively determined to be bad, rules by which the suspicious portions of the data are removed can be invoked. Owing to canonicalization, however, effective sanitization is extremely hard to implement. That is why sanitization should be used as a backup for rejection, and not vice versa.

5.3.3 Notification of Correct Input

It is reassuring to users when the application returns a notification message when the user's input has been accepted by the application. This is true whether the application itself, or a backend server, is

responsible for the final validation and disposition (acceptance, rejection) of the input. Therefore, write the application so that users are always notified not only when input is rejected, but when input is accepted.

5.3.4 Validations to Perform

The following sections describe the types of input validations that Web applications should perform.

5.3.4.1 Type Checks

Check to ensure that the input is, in fact, a valid data string, not any other type of object. This includes verifying that the input string contains no inserted executable content or active content, such as Trojan horses, malicious code, metacode, metadata, or metacharacters, HTML, XML, JavaScript, shell script, streaming media (or any other type of active or executable content). Note that executables and active content are often added to valid arguments to cause buffer overflows. Similarly, perform type checking on URLs and pathnames to verify that they are in fact URL and pathnames and do not contain metadata or pointers to malicious code.

If possible, sanitize any input strings to strip them of metadata or other suspicious tags; otherwise discard the form that contains the suspicious input. Under no circumstances should the application execute active content or metacode embedded in data input strings.

5.3.4.2 Format and Syntax Checks

Verify that the data string conforms with defined formatting and syntax requirements for that type of input, and perform canonicalization checks as appropriate. See Section 5.4.3.1.3. For example, the only acceptable format for a form field intended to collect Social Security Numbers is nine numeric characters. Reject and discard all incorrectly formatted or syntax-violating data.

5.3.4.3 Parameter and Character Validity Checks

Verify that any parameters or other characters entered (including format parameters for routines that have formatting capabilities; see Section 5.3.4.5) have recognized, valid values. Reject and discard any parameters that have invalid values. Filter all arguments, and select only the characters that are appropriate for the function being performed. If an argument is submitted as the result of a user selection from a pull-down menu or check box, make sure the value provided by the user is in fact one of the legal values.

It cannot be stated too often that Web applications should not be written to assume that data sent between browser and server is trustworthy and has not been changed during transmission, unless the data were protected from disclosure and tampering by encryption (SSL/TLS), and possibly also a digital signature or hash.

5.3.4.3.1 Handling Special Characters

As shown in Table 5-3.1, the following characters act as special characters in HTML paragraphs and XML blocks. They should be checked for in user input from browsers and in server output to browsers.

Table 5-1: Special Characters in HTML and Their Purpose

<	Introduces a tag
&	Introduces a character entity or separates CGI parameters
>	Is treated as special by some browsers that assume it was a typo and that the author really meant to enter <
" "	When enclosing an attribute value, marks the beginning and end of the attribute value
' '	When enclosing an attribute value, marks the beginning and end of the attribute value (see note that follows)
/sp /tab /nl	Indicates the end of a URL
non-ASCII characters	Shows what is not allowed in URLs; all characters greater than 128 in ISO-8859-1 encoding are non-ASCII, and are not allowed in URLs
%	To be filtered out if used in code that contains parameters encoded with HTTP escape sequences that will be decoded on the server; for example, % should be filtered out if "%68%65%6C%6C%6F" becomes "hello" when displayed on the Web page
; () { } /nl	To be filtered out if enclosed by <SCRIPT> </SCRIPT> if text could possibly be inserted directly into the preexisting script tag
!	To be filtered out of server-side scripts that convert exclamation marks into double quotes in output

NOTE: Although the XML specification allows use of single quotes (' '), some XML parsers do not handle them correctly. To be safe, use only double quotes (" ") in XML. Also note that attribute values not enclosed in quotes turn white space characters, such as /sp (space) and /tab into special characters. These are not legal in XML, where they turn other characters into special characters as well. Do not use unquoted attributes in XML if they will include dynamically generated values.

5.3.4.3.1.1 Filtering Special Characters

One approach to handling special characters is to simply filter them as soon as they are input or before they are output. To filter special characters during input validation, simply omit characters to be filtered from the list of valid characters to be allowed. For example, the following filter, written in Perl, will accept only one special character, /sp:

```
# Accept only legal characters:
$summary =~ tr/A-Za-z0-9\ \.\:\/\/dc;
```

To filter the minimum number of characters, it may be preferable to use a subroutine that removes only those characters (rather than one that checks for only valid characters). Here is an example again in Perl:

```
sub remove_special_chars {
    local($s) = @_ ;
    $s =~ s/[\<\>\\"'\%\\;\(\)\&\+]/g ;
    return $s ;
}
# Sample use:
$data = &remove_special_chars($data) ;
```

5.3.4.3.1.2 Encoding Special Characters

An alternative to removing the special characters is to encode them in a way that removes their special meaning. One advantage of encoding (commonly called HTML encoding) over filtering is that it avoids data loss, which is a risk with filtering, and still allows special characters to be displayed on the Web page. The HTML, XML, and SGML specifications all explain the correct encoding of special characters, and specify the encoding character sets (*charsets*) that may be used. For example, the HTML specification lists all mnemonic names, decimal numbers, or hexadecimal to be used in HTML character encoding. RFC 2279 references several possible text encoding *charsets*.

Implementers of UTF-8 need to consider the security aspects of how they handle illegal UTF-8 sequences. It is conceivable that in some circumstances an attacker would be able to exploit an incautious UTF-8 parser by sending it an octet sequence that is not permitted by the UTF-8 syntax. A particularly subtle form of this attack could be carried out against a parser that performs security-critical validity checks against the UTF-8 encoded form of its input but that interprets certain illegal octet sequences as characters. For example, a parser might prohibit the NUL character when encoded as the single-octet sequence 00 but allow the illegal two-octet sequence C0 80 and interpret it as a NUL character. Another example might be a parser that prohibits the octet sequence 2F 2E 2E 2F (" / . . / ") yet permits the illegal octet sequence 2F C0 AE 2E 2F.

To avoid potential problems when the encoded characters are displayed by the browser, the encoding character set (*charset*) used internally by the server should match the *charset* used to encode output to the browser. If the server's internal *charset* is not ISO-8859-1, make sure that the alternative encodings (e.g., UTF-7, UTF-8) for special characters do not inadvertently slip into output sent to the browser.

The best way to prevent incompatibility between the server's internal encoding *charset* and its output *charset* is to first translate the characters internally to ISO 10646 (which uses the same character values as Unicode), then use either numeric (decimal or hexadecimal) or character entity (mnemonic) references to represent the characters in output.

NOTE: Hexadecimal encoding is not supported in SGML (ISO 8879); you must use decimal encoding instead.

Regardless of which *charset* you choose for the application, always specify the encoding *charset* that should be used by users when they return data to the Web application, by specifying this *charset* in the HTML using the *charset* parameter. If the encoding *charset* is not specified, the user may use an encoding scheme not expected by the application to include encoded special characters or metadata that point to malicious content, or to disguise malicious code before inserting it into input data. In addition, unless older HTML 1.0 browsers must be supported or the programming library does not allow it, set character encoding as part of the HTTP protocol output, which would enable the server to send the desired *charset* value as part of the HTTP protocol.

5.3.4.3.1.3 The Canonicalization Problem

Canonicalization is the method by which systems convert data from one form to another. The canonical form of the data is the simplest, most standard form of that data. Canonicalization means the conversion of something from a more complex representation to its simplest form. Web applications perform canonicalization when doing URL encoding or IP address translation, among other activities.

The canonicalization problem arises when security decisions are based on canonical forms of data but the application is unable to accurately map, encode, and decode the canonicalized data.

5.3.4.3.2 *Handling Metacharacters and Metacode*

Metacharacters affect the behavior of programming language commands, operating system commands, individual program procedures, and database queries. They may be nonprintable or printable. None of the following characters, as shown in Table 5-3.2 should be accepted as legitimate input to Web applications.

Table 5-2: Metacharacters and Their Actions

<i>;</i>	Causes additional command execution
<i> </i>	Causes command execution
<i>!</i>	Calls to a command, which is then executed
<i>&</i>	Causes command-execution
<i>x20</i>	White space, can be used to fake URLs and other names
<i>x00</i>	Null bytes, to truncate strings and filenames
<i>x04</i>	<i>EOT</i> , for inserting an “end of file” indicator
<i>x0a</i>	<i>Newline</i> , to indicate additional command to be executed
<i>x0d</i>	<i>Newline</i> , to indicate additional command to be executed
<i>x1b</i>	Escape
<i>x08</i>	Backspace
<i>x7f</i>	Delete
<i>~</i>	
<i>' "</i>	Often combined with database queries
<i>-</i>	Combined with database queries and creation of negative numbers

*%	Combined with database queries
`	Causes command execution
/ \	Used to fake pathnames and queries and to insert scripting language-related tags in documents on Web servers
< >	Causes file operation
?	Programming/scripting language related
\$	Programming/scripting language related
@	Programming/scripting language related
:	Programming/scripting language related
() { } []	Programming/scripting language and regex related

5.3.4.3.2.1 Shell Metacharacters

Applications that will run on UNIX systems should particularly avoid including UNIX command shell (e.g., */bin/sh*) special characters, which are interpreted by the shell as metacharacters unless preceded by an *escap*” character. These special shell characters area as follows:

```

&      ;      `      '      \      "      |
*      ?      ~      <      >      ^      (
)      [      ]      .      =      {      }
$      !      -      --     #      [n]     [r]
[t]    [v]    [f]    [sp]  [nil]

```

In shell scripts, the default separators for parameters—`[t]` (*tab*), `[sp]` (*space*), and `[n]` (*newline*)—should be changed to different values, via the internal field separator (IFS) environment variable. In addition, if the source of any shell script is untrustworthy, the IFS environment variable should be discarded or reset during environment variable processing. Note that in both in UNIX shell and Perl, the back tick (```) also calls a command shell.

Shell metacharacters are pervasive because several important library calls in C and C++ are implemented by calling the UNIX command shell (and are thus affected by shell metacharacters). Avoid the following calls when writing code that spawns a process; use `execve ()` instead:

- `popen ()`
- `system ()`
- `execlp ()`
- `execvp ()`

5.3.4.3.2.2 SQL Metacharacters

SQL also includes metacharacters. Avoid including metacharacters in any program that calls to SQL. When formulating a SQL command, allow only a very limited pattern (format) for input, and allow only data that match this strictly defined pattern to enter the program. Limit the input pattern for SQL commands to one of the following:

```
^[0-9]$\
```

-or-

`^[0-9A-Za-z]*$`

If the application must handle data that may include SQL metacharacters, the data should be encoded before they are stored. For example, the application could encode the data—including ampersands (&—into HTML, with all user inputs (including numeric inputs) enclosed by quotation marks (" "). Refer to Section 5.3.4.3.1, Handling Special Characters.

5.3.4.3.2.3 File Disclosure Vulnerabilities and Path Traversal Attacks

If the Web application uses the Web server file system to temporarily or permanently save information such as image files, static HTML files, or CGI scripts, be aware that the *WWW-ROOT* directory is the virtual root directory within many Web servers. This directory is accessible by HTTP clients. The Web applications may store data within or outside, or both of *WWW-ROOT* in designated locations.

If the application does not properly check and handle pathname metacharacters such as `.. /`, the application may be vulnerable to a path transversal attack in which the attacker constructs a malicious request to return data about physical file locations, such as `etc/passwd`. Attackers may create specially crafted URLs to cause path traversal attacks in conjunction with other attacks, such as SQL injection attacks. Path traversal attacks may be used to traverse to system directories containing binary executables, thus enabling the attacker to execute system commands outside of the designated pathnames.

Preventing path traversals and path disclosures can be difficult in large distributed Web systems consisting of several applications. The architecture of such distributed systems should include a central point at which all requests are received and from which all requests leave, so that a single common security component can be used to detect and prevent path traversals and disclosures in those requests.

5.3.4.3.3 *Null Byte Checks*

Even if you write the Web application in a language other than C or C++, at some point in the application operation you will likely have the program pass data for further processing to an underlying low-level function (e.g., a library routine) in C or C++. For this reason, you need to be aware that in C and C++ the null byte (`\0`) is the terminator for a string.

Applications that do not perform adequate input validation can be fooled into prematurely terminating a string (with unpredictable results) by a user's insertion of null bytes into critical parameters, for example, by URL encoding the null bytes (i.e., `%00`) in an HTTP QUERY_STRING).

Perl is very susceptible to null byte attacks when executing system calls, such as `open` and `stat`. So are the *File*, *RandomAccessFile*, and similar classes in Java. PHP is also susceptible if not configured to avoid such attacks.

If all requests come into and leave from a central location (e.g., Web portal), a single component can be used to detect and prevent null byte attacks.

5.3.4.4 Divide-by-Zero Checks

Check the value of any numeric argument submitted to a calculation to verify that the argument cannot possibly cause the calculation divide by 0 (which can result in a fatal error). Write all calculations performed by the application to specify their variables in a way that makes division by zero impossible. For example, to calculate a divided by b , do not write (or accept in third-party code):

`a / b`

because b might = 0, which would result in an attempt to divide a by 0. Instead, write (or modify the calculation to read) this:

`\IF (b <> 0 , a / b)`

That ensures the calculation will be performed if and only if b is less than or greater than 0, but not if b equals 0, thus eliminating the potential divide-by-zero problem.

5.3.4.5 Check for User Input to Formatting Routines

The application should not accept user input as parameters to formatting routines. This is because there are many ways a hacker can exploit a user-controlled format string, including

- Submitting a long formatting string to create a buffer overflow.
- Using conversion specifications that include unpassed parameters to enable insertion of unexpected data instead of values intended for formatting or printing, or to enable the hacker to overwrite near-arbitrary locations by specifying an alleged parameter that was not actually passed. In many cases, the results of such operations are sent back to the user, making this kind of attack an attractive way to reveal internal information about the stack or to circumvent stack protection systems such as StackGuard (see Appendix C).
- Creation of formats that produce unanticipated results, for example, by prepending or appending awkward data to a value.

Routines and operations that have formatting capabilities include the following C/C++ routines:

- `syslog()` (and other routines whose name contains `log`)
- `setproctitle`
- `printf()`, `sprintf()`, `snprintf()`, `fprintf()` (and others the `printf` family)
- functions whose names begin with `err-` or `warn-`.

In Python, the operation `%` is a formatting operation. Also be aware of any programs or libraries that define formatting functions by calling built-in routines to do additional processing, for example, *glib*'s `g_snprintf()`. An example of a construct, in C, that would be vulnerable to a formatting string attack is this:

```
printf(string_from_untrusted_user);
```

The following C construct will achieve the same functional result, but without the vulnerability:

```
printf("%s", string_from_untrusted_user);
```

The following guidelines will help you avoid formatting string vulnerabilities:

- To implement simple formatting strings in C, use a constant string or `fputs()`.
- Filter all user data using a filter that lists the valid characters for the particular format string being used.
- Use a compiler that issues warning messages when it detects insecure format string usages (*gcc* compilers issue such warnings).
- Limit any formatting string that includes a function call to implement a lookup for internationalization (e.g., `gettext` in C) to only values controlled by the program.

5.3.4.6 Check for Session Token to Prevent URL Manipulation

When a user clicks a link on an HTML page, the selection is translated by the browser into an HTTP URL request, which is then sent to the Web server application via HTTP *GET* or *POST*. As with HTML forms, URL selections by users can be intercepted and manipulated by hackers, who change parameter values in the URL by to construct request strings to perform other functions, such as hijacking an e-banking transaction request and using it to transfer unauthorized purposes.

When a parameters needs to be sent from browser to server, it should be accompanied by a valid session token. The application should validate the session token to ensure that it is indeed the valid token associated with the requested username or account. The application should reject all parameters for which the session token cannot be validated as coming from an authentic user who is authorized to act on the specific account indicated in the URL.

5.3.4.7 HTTP Header Checks

HTTP headers are prepended to HTTP requests and responses to pass control information between the browser and the Web server. Each HTTP header normally consists of a single line of ASCII code including a name and a value. Most Web applications ignore HTTP headers.

However, you may need to have your application inspect incoming HTTP headers—in which case, the often-stated truism stands true: HTTP headers that originate from browsers should not be trusted; such

headers may be intercepted in transit from browser to server by a hacker and modified. Therefore, never use a header that originates from a browser (e.g., a referrer header) as the basis for making a security decision.

If headers must be relied on by the server application, write it to trust only headers that originate from other servers, and implement additional security measures to protect the headers to be relied on from tampering. For example, if a header originates from a server in a cookie, encrypt, sign, and hash the cookie before sending it, and do the appropriate validations on the header it contains when receiving it.

Browsers themselves do not allow header modification. However, hackers can easily write their own small programs (as little as 15 lines of Perl) to perform the modifying HTTP request, or they may use one of several freely available proxies that enable easy modification of HTTP headers (and of all data sent from the browser). Two common types of HTTP header manipulation attacks are described below.

- *Example of referrer header tampering:* The referrer header is sent by most browsers and usually contains the URL of the Web page from which the HTTP request was sent. Some Web server applications are written to check this header to verify that the request originated from a Web page on that Web server, in the erroneous belief that if the HTTP request can be proven to originate from a page on the Web server itself, it cannot possibly have originated from a copy of the page downloaded, saved, and modified, and posted by the hacker on the hacker's own server. This verification check is pointless, however. A hacker resourceful enough to download and modify the Web page itself will be resourceful enough to modify HTTP referrer header to make it appear as if modified page were sent from the same Web server from which the original page was downloaded by the hacker.
- *Example of Accept-Language Header Tampering:* The Accept-Language header indicates the preferred human languages of the user. A Web application that performs internationalization may read the language label in the HTTP header and pass it to a database as a pointer to look up a text in the preferred language. If the content of the header is sent verbatim to the database, an attacker may be able to embed SQL commands in that (an attack known as SQL injection; see Section 5.4.1.2). Similarly, if the header content is used to build the name of a file from which to look up the correct language text, the hacker may use it to launch a path traversal attack.

5.3.5 Virus Scanning

The best way to handle virus scanning of HTTP, SMTP, and FTP content is not from within the server application (e.g., through a call to a virus scanner), but by deploying a dedicated virus scanning system to scan all data before they are even allowed to reach the server application. For example, a virus scanning gateway, such as Trend Micro's VirusWall (see Appendix C) could be in the network DMZ with the firewall. The reason that virus scanning should not be implemented by (or from within) the application is that virus scanning is a very high-overhead process that uses significant CPU resources.

Thus, it should not even be implemented on the Web server platform with the application but should, as indicated, be relegated to its own dedicated platform.

5.4 INTEGRITY AND INPUT VALIDATION IN DATABASE APPLICATIONS

For best performance, define and enable integrity constraints in the backend database used by the Web application, and develop the application to rely on those constraints rather than on SQL statements to enforce the database's business rules.

That said, in some cases, you might want to enforce business rules through both the application and the integrity constraints. A first level of enforcement by the application may more quickly feed back to the user than with use of an integrity constraint alone.

For example, consider an application that is designed to accept 25 values from a user, and to use those values to construct and submit an SQL *INSERT* statement to the database. The database's integrity constraints can be enforced only after all 25 values have been entered by the user and the application has submitted the SQL *INSERT* statement containing those values. If any of the values violates a business rule, the user will not be notified until he has finished submitting all 25 values and the SQL statement has been received and validated by the database.

By contrast, if the application itself performed a first-line validation of each value as the user entered it, and returned an error message immediately if any value was found to be in violation of business rules, the user could correct the value and resubmit it immediately, without having to wait until he had entered all 25 values and the application had submitted them in a SQL statement. This makes the processing of database requests much more efficient and also prescreens values before they reach the database, so that the database's integrity constraints are used for exception handling of only those values that may have got past the application's integrity checks.

5.4.1.1 Reparsing Requests and Data for Backend Databases

The application that transforms HTML forms into SQL requests must keep the data input in those HTML forms secure and correct during the conversion process, and during the transmission to the backend database. Most major database management systems provide software packages, libraries, and APIs that enable Web applications to translate HTML forms requests into SQL queries. For example, Oracle provides Java and PL/SQL Web interface solutions.

5.4.1.2 Avoiding Direct SQL Injection

Direct SQL injection is an attack that enables malicious users to make direct SQL calls to a backend database by manipulating—creating or altering—SQL commands transmitted in the browser's HTTP requests to the server. SQL injection is used to gain access to data for which the attacker is not authorized. This attack succeeds on systems in which input validation routines do not exist or are poorly designed.

Too many Web applications assume that an SQL query is a trusted command. Thus, SQL queries are able to circumvent Web server access controls and bypass standard authentication and authorization checks. In some instances, attackers can use SQL queries to gain access to operating system commands. Direct SQL injection attacks are also used by hackers to

- Change SQL values
- Concatenate SQL statements
- Add function calls and stored-procedures to a statement
- Typecast and concatenate retrieved data.

To prevent SQL injection attacks, take these measures:

1. Validate all user input to SQL queries, and accept only expected data types.
2. Validate and sanitize every user variable passed to the database.
3. Quote user input passed on to the database.
4. Use user-supplied data to build SQL read-only queries only, not to update databases.
5. Never accept direct SQL queries (or query fragments) as input data.
6. Filter out any special characters from SQL statements, including + ' = .

5.4.1.2.1.1 Example of a Direct SQL Injection Attack

A Web application includes functionality that enables users to change their passwords. To do so, the server presents an HTML form to the user with four blank fields:

```
Username:
Old password:
New password:
Confirm new password:
```

When the user enters the requested information in the HTML form fields, the browser translates the supplied data into an HTTP request, which it sends to the server application:

```
http://www.server.mil/changepwd?pwd=O!dP@sswD&newpwd
=5Q1!nject&newconfirmpwd=5Q1!nject&uid=testuser
```

The server application extracts the four parameters from the HTTP request:

```
Pwd=O!dP@sswD
Newpwd=5Q1!nject
Newconfirmpwd=5Q1!nject
Uid=testuser
```

It then checks to make sure that *Newpwd* matches *Newconfirmpwd*. After verifying that the two match, the application discards *Newconfirmpwd* and builds an SQL query, which it sends to the password database, to validate the old password and then overwrite it with the new password:

```
UPDATE usertable SET pwd='$INPUT[pwd] ' WHERE
uid='$INPUT[uid] ' ;
```

A hacker who knows how this process works will also know he can insert an additional database function within the valid SQL request. He could modify the HTTP request generated by the browser (before that request is transmitted to the server), to include an additional function that replaces the password of all accounts named *admin* with his own password. Here is an example:

```
http://www.server.mil/changepwd?pwd=0!dP@sswD&newpwd
=5Q1!nject&newconfirmpwd=5Q1!nject=testuser
'+or+uid+like'%25admin%25';--%00
```

This modified HTTP request is translated by the server into an SQL query that instructs the password database to reset the administrator password to the hacker's password. The result is that the hacker gains unlimited access to the Web server, whereas the legitimate administrators are locked out.

5.4.2 Validate Originators of Data and HTML

Usually Web applications implement only one-way authentication (via SSL/TLS) to assure the client that it is indeed connected to the expected server. However, SSL does support two-way authentication, whereby client and server mutually authenticate one another, so the server is also assured that it is connected to the expected, known client. Two-way authentication using SSL should be used to prevent unauthorized modifications of data on the Web server by unauthorized clients.

For example, a browser connects to the Web application server (e.g., J2EE) via SSL to submit a Web form (HTML, JavaScript, etc.) containing a request for data. SSL secures all data in transit between browser and server, whereas the server itself must protect the data it receives from the browser, using all of its standard access controls and other security mechanisms.

The form data could then be translated by a server process into an XML-based query and passed to a data access service running in the Web application server. The data access service would then translate the XML query into the appropriate SQL calls to one or more RDBMSs. The responses could then be returned via an XML response to the application server, which could then be rendered into whatever format the original browser request was received in (e.g., HTML, JavaScript).

NOTE: This is the approach used by the Navy's Task Force Web (TFWeb) Enterprise Portal.

5.5 APPLICATION AWARENESS OF THE OPERATING ENVIRONMENT

We strongly recommend using a Web application server platform that provides environment data to applications that run on the server. For example, J2EE server components run within contexts, such as, System Context, Log-in Context, Session Context, and Naming and Directory Context, etc. These contexts provide environment data to the application at run time.

5.6 PROTECTING APPLICATION CONFIGURATION DATA

Make sure that only the application and the administrator are authorized to access the configuration data. Use the system-level access controls to protect configuration data stored locally with the application, assigning write and delete privileges only to the system administrator, and read privileges only to the administrator and the application itself. If the system-level access controls alone are considered insufficient to protect the configuration file, store the file in encrypted form, and write the application to decrypt the file whenever it needs to reference it (e.g., at start-up).

Another alternative is to store the application's configuration information remotely, such as in the same directory from which the application retrieves cryptokeys and certificates for authenticating users. In this way, the application can retrieve its configuration information using LDAP, and it can benefit from the security protections of the directory server which, because it stores security data and cryptographic, may be more strongly protected than the application's own server.

If the configuration file are retrieved from a remote directory (or other remote database and server), the configuration data should be transmitted from directory to application via a channel-encrypted SSL/TLS.

Depending on the application's susceptibility to being cloned, it may be necessary to protect the configuration data from ever being copied directly from the application. The simplest approach would be to prohibit the configuration data from being transmitted back across the communication channel over which it was received. If readback verification of the configuration received by the application is required, use encryption to prevent the data from being copied in transit.

5.7 INTERPROCESS AUTHENTICATION: BEYOND CHALLENGE AND RESPONSE

5.7.1 Kerberos

Kerberos is not only a useful technology for implementing SSO, but also for supporting interprocess authentication. Indeed, because it is widely available, understood, and tested, Kerberos represents an effective alternative for implementing interprocess authentication at this time, particularly in DoD applications.

5.7.2 X.509

At this time, interprocess authentication using X.509 certificates is unworkable. The X.509 certificate management infrastructure is not designed in a way that it can easily accommodate the on-the-fly delegation of user identity certificates to processes operating on a user's behalf, nor easily accommodate the need to rapidly expire certificates used as interprocess authentication tokens.

That said, X.509-based (as well as other non-Kerberos) interprocess authentication schemes and infrastructures have been proposed. A handful are in development, mainly by organizations involved in supercomputing. Significant is that being developed by the Globus Project as part of its Grid Security Infrastructure (GSI). The GSI provides an SSO, run-anywhere authentication service based on SSL and X.509. It has support for local control over access rights and mapping from global to local user identities, for applications operating in the Globus High-Performance Computer Grid. For more information, see <http://www.globus.org/security/>.

5.7.3 Secure Remote Procedure Call

Remote procedure call (RPC) specifications are limited to relatively narrow applications that are confined within a single administrative domain. Since many Web applications need to operate across domain boundaries, RPC for Web applications needs a comprehensive security infrastructure beyond what is possible by simply layering the RPC mechanism over SSL/TLS.

The OpenGroup DCE provides for use of authenticated RPCs between clients and servers. Authenticated RPC works with the authentication and authorization services provided by the DCE Security Service, specified in the RPC run-time library for the particular server application for which authenticated RPC is being enabled. DCE specifies a number of authenticated RPC routines that can be used by client-server application programmers in this context.

For more information on authenticated RPC in DCE, see http://www.transarc.ibm.com/Library/documentation/dce/1.1/app_gd_core_15.html#14.2

5.8 USE OF MOBILE CODE

5.8.1 Use Only Approved Mobile Code Technologies

Section 4.8 of the *Recommended Application Security Requirements* document defines and lists the different categories of mobile code that may be used in DoD Web applications, as well as the circumstances under which they may be used (that is, must be signed versus. no signature required).

The safest approach to mobile code use would be to avoid using any Category 1 and Category 2 mobile code and to limit any mobile code to Category 3 mobile code, such as JavaScript and PDF (the latter used for document downloads; convert all Postscript documents to PDF before posting on the Web server). Ideally, the client will be exposed only to HTML and JavaScript (the latter used for trees,

forms integration, data validation, etc.) served from a J2EE Application Server via Java Server Pages (JSP)/servlets.

Servlets offer some important benefits over earlier dynamic content generation technologies. Servlets are compiled Java classes, so they are generally faster than CGI programs or server-side scripts. Servlets are safer than extension libraries, because the Java Virtual Machine (JVM) can recover from a servlet that exits unexpectedly. Servlets are portable both at the source-code level (because of the Java Servlet specification) and at the binary level (because of the innate portability of Java bytecode).

If Java applets are used, because they fall into Category 2 mobile code, they must be digitally signed by the server before they are served to the browser. DSA or SHA-1 are the DoD-mandated hash algorithms that should be used to digitally sign mobile code. Java's Cryptographic Service Provider supports validation of SHA-1 digital signatures.

5.8.2 Mobile Code Signature and Validation

Signature of application code, including mobile code signature, was discussed in Section 4.5.3.1.

5.8.3 Secure Distribution of Mobile Code

Beyond code signing and validation of code signatures to determine, after the fact, whether openly transported mobile code has been tampered with in transit, very little attention has yet been devoted to the secure distribution of mobile code programs. That pertains to secure distribution techniques that are designed to prevent (rather than detect) code tampering, as well as misrouting of mobile code, illicit copying, and the like.

The currently predominant commercial model for mobile code distribution identifies dynamically linkable parts of mobile programs by a URL. This model assumes that all of the constituent parts that make up a mobile program will be downloaded to a single location, where they will be verified, linked, possibly dynamically compiled, and finally executed at that same location. Besides the obvious defects of a distribution management and versioning scheme based on untrustworthy URLs, this fairly limited, simplistic distribution model is neither flexible or scalable enough to support other modes of mobile-code dissemination and deployment. The NSA identified this problem with regard to Java application distribution, and proposed an architectural solution, in the 1996 issue of *Government Information Technology Issues* (<http://www.gitec.org/assets/pdfs/pdf96/robins.pdf>). It is not clear whether NSA ever pursued this work beyond the laboratory research stage.

The Navy-Marine Corps Intranet (NMCI) implements a process to approve mobile code before adding it to the Gold Load (final version of software for installation). Any NMCI browser that needs a particular piece of mobile code that is not already loaded locally (via the Gold Load) can go to an approved trusted site and download the new digitally signed piece of mobile code via an SSL pipe that has been mutually authenticated through DoD Class 3 certificates.

Other approaches to the problem are being defined and implemented by R&D projects in the United States and Europe. The Transprose Project sponsored jointly by the Office of Naval Research and

DoD Critical Infrastructure Protection and High Confidence, Adaptable Software (CIP/SW) Research Program is defining a flexible, scalable, and secure mobile code distribution model and extensible architecture for mobile code management and secure distribution that will accommodate any current or future method of mobile code distribution (see <http://nil.ics.uci.edu/transprose/> and <http://www.ics.uci.edu/~franz/CIP.html>). On a smaller scale, more immediately pragmatic approach, the European ESPRIT Project has developed FILIGRANE, a cryptography and smart card based system for securing mobile code distribution (see <http://www.dice.ucl.ac.be/crypto/filigrane/>).

The design of any DoD Web application system that incorporates the use of mobile code needs to address the security of the distribution of that code. The previous examples are provided to help stimulate your imagination as you undertake the definition of your own approach to solving the secure distribution of mobile code within your specific Web application.

6.0 MAKING APPLICATIONS RESISTANT TO INTERNAL FAILURE

Availability of applications becomes a security issue when that availability can be threatened by an outside attacker, in the form of a denial of service attack, or when an application failure can cause or enable other security violations, such as a failure that invalidates the access controls that protect sensitive information. Application susceptibility to denial of service and other availability-related violations is addressed to a great extent by the input validation measures described in Section 5.3. But input validation alone will not ensure application availability.

Applications may be made vulnerable to externally induced failures due to faults in their design, logic errors and bugs in their code, and inadequate or incorrect handling of errors and exceptions. If an attacker discovers that an application is vulnerable because of any of these problems, he can exploit that vulnerability to create a denial of service in the application or to bypass its security protections.

Correctness of the application design and implementation and effectiveness of its error and exception handling are critical to application availability in general, and resistance to denial of service in particular. Secure application design was discussed in Section 3.2.9. Writing of elegant, error-free code and avoidance of logic errors were discussed in Section 3.2.9.14. This section will deal with controlling application operation in ways that will minimize susceptibility to failure, and correct error and exception handling and recovery.

6.1 CONTROLLING OPERATION AND AVAILABILITY

6.1.1 Availability Requirements for DoD Applications

Although the availability of all DoD applications is important, the requirement for availability of both clients and servers in Mission Category 1 and other high-priority applications is imperative. These applications must be designed, coded, and tested with the utmost care to eliminate software errors that could cause the application to crash, as well as exploitable vulnerabilities that expose the application to a denial of service attack.

Mission Category in this context does not mean Defense Mission Category (DMC, as listed in Part III, Supplemental Codes, of *Army Management Structure, FY2002* [DFAS-IN 37-100-02]). Rather, regarding information systems, it is defined in the DoD Chief Information Officer (CIO) document *Public Key Enabling (PKE) of Applications, Web Servers, and Networks for the Department of Defense (DoD)* (17 May 2001, Attachment–Definitions pp. 9–10).

***Mission Category:** Applicable to information systems, the category reflects the importance of information handled by the information system relative to the achievement of DoD goals and objectives, particularly the warfighter's combat mission.*

The document defines *three priority levels for information systems: mission critical, mission support, and administrative*. The information system mission categories all fall under the mission critical priority level:

1. *Mission critical (high priority)*: Systems handling information determined to be vital to the operational readiness or mission effectiveness of deployed and contingency forces in terms of content and timeliness. It must be absolutely accurate and available on demand (may include classified information in a traditional context, as well as sensitive and unclassified information). Mission critical systems include the following mission categories of systems:
 - a. *Mission Category 1*: Defined by the Clinger/Cohen Act as National Security Systems (NSS), systems used to perform intelligence activities; cryptologic activities related to national security; systems used to perform command and control of military forces, systems integral to a weapon or weapons system; systems critical to direct fulfillment of military or intelligence missions
 - b. *Mission Category 2*: Direct mission support systems identified by the Commanders in Chief (CINCs) which, if not functional, would preclude the CINC from conducting missions across the full spectrum of operations, including readiness (to include personnel management critical to readiness), transportation, sustainment, modernization, surveillance/reconnaissance, financial, security, safety, health, information warfare, information security, and contractual;
 - c. *Mission Category 3*: Systems required to perform department-level and component-level core functions.
2. *Mission support (medium priority)*: Systems handling information that is important to the support of deployed and contingency forces. This information must be absolutely accurate but can sustain minimal delay without seriously affecting operational readiness or mission effectiveness (may be classified, but is more likely to be sensitive or unclassified).
3. *Administrative (basic priority)*: Systems handling information that is necessary for the conduct of the day-to-day business, but which does not materially affect support to deployed forces or the readiness of contingency forces in the short term (may be classified, but is more likely to be sensitive or unclassified).

As a developer, you must make every effort to ensure that applications that fall into Mission Category 1 under the mission critical priority are as error free, bug free, and vulnerability free as humanly possible.

Note: Because of the need for such high reliability in Mission Category 1 systems, it is very unlikely that Web technologies will be used in Mission Category 1 systems, since it is difficult to provide the required assurances about the reliability of Web technology.

The Web developer must be able to take for granted that the Web server and the operating system and hardware platform on which it runs satisfy the reliability requirements for the priority and mission category of system for which they were procured.

Nevertheless, there are availability issues that are specific to Web applications. Some of these were already addressed in the context of Web application integrity because in some cases, the same application vulnerability can be exploited either to attack integrity or to instigate a denial of service. For example, buffer overflows may be used to create stack overflows so that malicious code may be loaded—an application integrity issue—or they may be used to cause the system to crash—an application availability issue.

6.1.2 Input Time-Outs and Load Level Limits

Place time-outs and load level limits, especially on incoming network data accepted by the application, to prevent attackers from launching flooding denial of service attacks.

In addition, we strongly recommend using only a Web server that can implement some form of load shedding or load limiting to handle excessive load without crashing. For example, the Web server should have a configurable incoming request threshold so that if it detects an unusually large flood of incoming requests, it will stop processing all incoming requests once the load reaches that predefined threshold. Network daemons should also be configured to shed or limit excessive loads. An example is by setting values (using `setrlimit()`, if running on UNIX) to limit the resources that will be used by the daemon.

The Web server should also be configured with reasonable time-outs on the real time used by any process; once this time-out threshold is reached, the process should clean up the resources allocated to it by the server and exit. This will prevent the server from bogging down due to blocked processes, such as hung read requests from remote servers or browser rejection of data returned to it by the server.

Similarly, the server should configure reasonable limits on the CPU time allotted to any process to prevent bugs in the process code from putting that process into an infinite loop.

6.1.3 Adjusting to Unresponsive Output

If a browser is halted or its TCP/IP channel response is slowed, the application should be able to adjust by releasing locks quickly before replying to prevent the possibility of a denial-of-service attack. Always configure time-outs for anticipated user responses to outgoing write requests, to prevent those requests from causing the server to hang if the client does not respond in a timely manner.

6.1.4 Preventing Race Conditions

6.1.4.1 What is a Race Condition?

A race condition is an anomaly caused by a process's unexpected critical dependence on the relative timing of events. Race conditions usually arise when processes attempt to access a shared resource

(such as a file or variable), and this multiple access has not been anticipated by the developer and properly controlled.

Most processes do not execute atomically. They are subject to interruption between instructions by other processes. If a process is not prepared to handle such interruptions, another process may be able to interfere with it. Use this rule of thumb: Any given pair of operations in a secure program must work correctly regardless of how much of another process's code is executed between those operations.

The problem of failing to perform atomic actions repeatedly comes up in the file system. In general, the file system is a shared resource used by many programs, and some programs may interfere with its use by other programs. Race conditions fall into two categories:

1. Interference caused by untrusted processes (known as a sequence or nonatomic condition). This kind of interference is caused by processes that run other programs that are able to insert actions between operations of the secure program. An attacker may invoke such untrusted processes specifically to cause a race condition.
2. Interference caused by two or more different trusted processes (known as deadlock, livelock, or locking failure conditions) attempting to run the same program at the same time. Because these different trusted processes may all have the same privileges, they may be able to interfere with each other in ways that other programs cannot. An attacker may be able to exploit this type of interference.

6.1.4.2 Preventing Deadlocks

There are often situations in which a program must ensure that it has exclusive rights to a resource (file, device, server process). Depending on how the application is designed, multiple copies of a component program may run simultaneously. A deadlock can occur if these programs are stuck waiting for each other to release resources. For example, if one program attempts to lock File A and File B at the same time another program is already holding a lock for File B and attempting to lock File A, a deadlock will occur.

6.1.4.2.1 Using Files as Locks

To avoid deadlocks, implement file locking for any files that are modified during application execution. Provide a method for recovering the file locks if the program crashes while the lock is held. On UNIX systems, this can be done by creating a new file to indicate a lock, and programming all processes to cooperate with the new file or file locks.

The correct way to create a lock for processes on a local UNIX file system is to `open ()` the file with the following flags set: `O_WRONLY / O_CREAT / O_EXCL` and with no permissions (the latter prevents other processes with the same owner from obtaining the lock). `O_EXCL`, which is the flag for creating exclusive files in UNIX, works even for root files.

If the system is running Network File System (NFS), be sure it has been updated to the latest version, NFS Version 4. Previous versions did not completely support the semantics necessary for designating files as local to the client.

All programs that perform file locking must cooperate; noncooperating programs must not be allowed to interfere with other programs' cooperative file locking. Finally, the directories being used to store file locks must not have file permissions that allow any process except the file's owner process to create or remove file locks..

The Filesystem Hierarchy Standard (FHS 1997) referenced by Linux and some UNIX systems describes standard conventions for locking files, including naming, placement, and standard contents of these files. To ensure that the server application does not execute more than once on a given machine, create the server's process identifier to be */var/run/processname.pid* with the *pid* as its content. Also, place device lock files in */var/lock*. This approach has the minor disadvantage of leaving files hanging around if the program suddenly halts, but it is standard practice and that problem is easily handled by other system tools.

Programs that cooperate in using files to represent file locks must use the same directory, not just the same directory name, to store those lock files. Also, locks that should only work locally (on a single machine) should be stored in an unshareable directory. Locks that should be obeyed by programs distributed across separate machines should be stored in a shareable directory. The FHS explicitly states that */var/run* and */var/lock* cannot be shared, whereas */var/mail* can be.

6.1.4.2.2 Other Approaches to Locking

On network servers, the act of binding to a port acts as a kind of lock for the simple reason that if an existing server is bound to a given port, no other server will be able to bind to that port.

Another approach to locking for UNIX applications is to use POSIX (portable operating system interface) record locks, implemented through `fcntl()` as discretionary locks. Because they are discretionary, use of these locks requires all programs that need the locks to cooperate as with use of lock files. POSIX record locking is mandated in the POSIX 1 standard, and it is supported on virtually all UNIX and Linux systems. It can be used to lock a whole file or only portions of that file, and it can differentiate between read locks and write locks. If a process dies, its locks are automatically removed.

Mandatory locks, based on the UNIX System V mandatory locking scheme, can be used for files that have their *setgid* bit set but that do not have their *group execute* bit set. Before a mandatory file lock can be applied, the file system must be mounted. Once the lock is applied, every `read()` and `write()` is checked for locking, which makes mandatory locks more thorough than advisory locks. But it also makes them more resource intensive. Mandatory locks are available on UNIX System V and its derivatives, and Linux systems, but not necessarily on other UNIX versions. Finally, because processes with root privileges can be held up by a mandatory lock, a hacker might exploit its use to launch a denial-of-service attack.

6.1.4.3 Preventing Sequence Conditions

Application programs can be interrupted between operations to allow another program to run. This other program may be a malicious program that is designed to abuse or subvert the interrupted program. Carefully check all code to identify any pairs of operations that may fail if arbitrary code were to execute between them.

The processes for loading and saving a shared variable are usually implemented as separate, nonatomic operations: the increment variable operation is usually converted into a loading- and incrementing-saving operation. For this reason, if the variable's memory is shared with another process, this sharing may interfere with the incrementing.

Write secure programs to verify that a request should be granted, and if so, to act on that request. Do not provide any way for an untrusted user to change any of the criteria checked by the programs in making this determination before the program is able to act on the determination. This kind of race condition is sometimes called a time of check/time of use (TOCTOU) condition.

In particular, when the program performs a series of operations on a file, such as changing its owner, status, or permissions, the file should be opened in a way that will prevent the file from being overwritten while the program is running.

Create files (and directories securely). For example, when creating temporary files in UNIX, use `mkstemp()`, not `mktemp()`. Assign the newly created file only a minimal set of privileges; later expand those privileges as necessary. Never set the world readable permission on a file, for this could grant an attacker privileges he should not have.

Immediately after the file is created, invoke the exit handler or use the file system's semantics to unlink the file so that the file's directory entry disappears while the file itself remains accessible until the last file descriptor pointing to it is closed. In the application, the file should be accessed by passing the file's descriptor. Unlinking the file ensures that the file will be automatically deleted if the program crashes.

NOTE: Administrators may find it harder to determine how disk space is being used for they will not be able to display the file system by name.

Refer to *Secure Programming for Linux and UNIX HOWTO*, "6.10. Avoid Race Conditions," for extensive information and code examples on avoiding race conditions in UNIX applications.

6.1.4.3.1 Temporary Files and Race Conditions

A common trick by hackers is to create symbolic links in the file system's temporary (`/tmp`) directory to another file, such as the password file, while a secure program is running. The objective of this attack is to fool an executing secure program into symbolically linking to another file when it discovers that a given filename does not exist, and performing the same operation (`open`) on the linked file.

Do not simply reuse temporary filenames; remove and recreate them. This will prevent a hacker from observing the original filename and hijacking it before it can be reused legitimately.

Also, if possible use environment variables in the file system to enable temporary files to be moved out of the shareable */tmp* directory into a nonshared user directory accessible only to the application and its users. If available, implement a temporary file directory policy in the file system (e.g., *openwall* on Linux) to prevent processes from making hard links to files to which they have no write access, and to prevent root processes from following symbolic links that are not owned by root.

6.1.5 Application Invocation of Backup

We strongly recommend use of a Web and application server that provides a scheduling service that can drive regular data archiving of the server files.

6.2 ERROR AND EXCEPTION HANDLING AND RECOVERY

The application should include an error/exception handling and recovery mechanism with capabilities as will be discussed:

6.2.1 Failing Safe

Applications must be able to gracefully terminate. On termination, they must systematically and completely deallocate all resources they have used and delete all temporary data they have produced during their execution. In addition, to be considered secure, an application should always fail safe.

Fail safe means the application is designed so that if it does fail, the application rejects (does not respond to) any subsequent requests or inputs. The application must not be allowed to simply recover when a serious error or violation occurs, for this may cause the application to enter a state that threatens system security.

In security-critical applications, any misbehavior is detected when the application is processing a request (e.g., malformed input, a can't-get-here state, etc.). That should cause the application to immediately deny service and stop processing that request. Although such process termination may be manifest to the user as decreased reliability or usability, it will reduce the likelihood of errors causing exploitable security vulnerabilities.

When the application stops processing a request, the application should not terminate altogether. This is particularly true of server applications; if such applications shut down every time they receive malformed input, they will soon be recognized as easy targets for denial of service attacks. On the other hand, if the malformed input creates a critical condition, such as a can't-get-here state, failing safe—i.e., complete application termination while maintaining a safe system state—may be the only acceptable alternative. Application error handling should be designed so that different criticalities of errors are clearly defined and recognizable by the error handling mechanism and so that only the most critical errors cause a complete application termination.

The only applications in which the previously defined process termination may be unacceptable are mission-critical applications, those in which the need for availability outweighs the need to protect data confidentiality or integrity. Web applications, no matter how important, would not be designated mission critical. There is not a high degree of assured availability required for mission-critical applications.

6.2.2 Error Detection

The application should contain an error detection capability, either one embedded in the application or an external detection facility linked to the application, that detects failure events, processing errors and exceptions, and possible security violations. This capability should support appropriate levels of alarms to alert the administrator of these events, errors, and violations. The capability may allow the administrator to choose which events should trigger these alarms, or it may simply generate notifications for all security-relevant events. All output from this error detection capability should be integrated into the application audit trail.

6.2.3 Resistance to Denial of Service

The application's error-handling mechanism must resist denial of service attacks in which the hacker floods the application with multiple malformed arguments (or other data) in an attempt to cause the error and exception-handling mechanism to consume so many resources that the application no longer has enough resources to continue processing and becomes unavailable to legitimate users.

6.2.4 Administrator-Configurable Error Responses

The error-handling mechanisms should enable the administrator to configure the application's responses to various errors and failures, providing at a minimum the following options for graceful termination:

- Entire application terminates
- Erroneous process only terminates
- Erroneous process and other selected processes terminate
- Termination triggers user notification: *YES* or *NO*
- Termination triggers administrator notification: *YES* or *NO*
- Termination triggers automatic checkpoint restart: *YES* or *NO*.

6.2.5 Transaction Rollback and Checkpoint Restart

If the application is transaction oriented, such as a database application, it should include a checkpoint restart capability that allows allow transaction rollback after failure. Transaction rollback means that the transaction is able to resume processing at the point just before it failed.

To ensure that rollback gets rolled forward to the Web application

- We strongly recommend using a Web server and application server that provides a transaction service (standard or add-on).
- Program the application to use that transaction service

For example, in the J2EE application server, a condition in the backend database that results in the rollback of a transaction will cause an exception to be thrown in the J2EE server. The server's standard exception-handling mechanisms can be easily used to notify the user, such as by displaying a resubmission screen.

6.2.6 Consistency Checking Before Restart

The application should include consistency-checking code that verifies that the validity of the application's call arguments and basic state assumptions before restart. For example, if a variable inside the program is meant to have only the values 1, 2, or 3, the consistence check will reveal whether it contains any other values, and it will generate an error condition if it does. For example, in C, macros such as `assert()` can be used accomplish consistency checking.

6.2.7 Safe Error Messages

Standard Web server error messages may disclose information that can be exploited by attackers, including file pathnames and system architectural details. For example, if an *include* file is not found by an executing application, it may return the following error message:

```
include file: c:\inetpub\wwwroot\common.asp not  
found
```

By including the pathname, the error message reveals to the attacker exploitable information about the Web server directory structure. There is no reason to include such information in the error message that is returned to the end user.

Similarly, attackers may infer characteristics of the directory structure from HTTP 301, 302, and 404 error messages. Therefore, instead of returning a generic HTTP error message when a user enters a nonexistent URL, the application should automatically redirect the user to a main index or home page associated with the topmost parent directory in the Web directory structure.

Error messages may also include unnecessary information about the Web server system architecture. For example, an Open Data Base Connectivity (ODBC) error message may reveal the brand name and release number of the DBMS used by the application. Other error messages have been known to contain the exact version of the underlying operating system or of the CGI scripting engine used by the developer. Such information can be exploited by attackers to target known vulnerabilities in a specific technology, such as a particular brand or version of the DBMS or operating system.

Even indirect information may be useful to an attacker. For example, an error message that reveals that the Web server runs on an older version of an operating system or uses an older version of a DBMS

signals to the attacker that the system as a whole may not be properly maintained, and is likely to contain exploitable vulnerabilities.

Detailed system architecture information in error messages can also be exploited by insiders in social engineering attacks, particularly in large organizations.

Error messages returned to the user when an application process (or the whole application) fails should reveal only a minimal amount of information, just enough to help the honest user understand the general cause of the failure, for example, “access denied” or “invalid input.” Error messages to users should not include file pathnames or system architecture information. But this kind of detailed information should be written to the application’s error log and audit trail, and it should be included in any alert messages sent to the administrator.

Be aware that some brands of Web servers, DBMSs, and other COTS servers default to issuing detailed error messages (Microsoft’s Internet Information Server, IIS, does this). The developer and administrator should both carefully review the error-related configuration details of the server used, as well as the ways errors are handled throughout the application. The server should always be reconfigured to issue uninformative error messages to users. For example, with IIS, the administrator can choose (in *Home Directory/Configuration/App Debugging*) between the default “Send detailed ASP error message to client” and the less informative “generic error”; the latter option should always be selected.

Before a Web site is allowed to go on-line, the development quality assurance team should systematically check all pages and scripts on the site to ensure that no detailed error messages are returned by any of them.

6.2.8 Error Logging

It is better to log too much than too little. In addition to writing the results of standard application errors to the Web server or DBMS error log, the application error-handling mechanism should write the same information to its own dedicated application log file. Or it should use the operating system’s log facility (e.g., UNIX *syslog*) to capture errors.

NOTE: If syslog is used by a UNIX application, the application should perform bounds checking on arguments before passing them to syslog() to avoid buffer overflows.

7.0 DEVELOPMENT TOOLS

7.1 APPLICATION MIDDLEWARE FRAMEWORKS

Application middleware frameworks are software packages that simplify the construction of distributed applications by providing prepackaged standard mechanisms, including security mechanisms and the APIs needed to integrate those mechanisms with application components.

The most significant benefits accrued from using application development middleware are:

- Reduced amount of required custom development
- Reduced risk of improper integration of application components
- Increased application interoperability with other applications
- Increased application extensibility, portability, and reusability.

Significant application middleware packages are the Distributed Computing Environment (DCE), the Distributed Component Object Model (DCOM), and the Common Object Request Broker Architecture (CORBA), and Java Remote Method Invocation (RMI) for Java applications (CORBA is also tightly integrated with Java and supportive of Java development).

7.1.1 Distributed Computing Environment

DCE, developed by the Open Software Foundation (now OpenGroup), is the oldest of these application middleware frameworks. It may still be useful for extending the capabilities of legacy applications built with DCE, but it should not be used in new Web application efforts. Many of the most innovative features of DCE have been incorporated into both DCOM and CORBA. Unlike DCE, these later middleware packages—as well as Java RMI—are implicitly and explicitly based on object-oriented methodologies.

In its *Recommendations for using DCE, DCOM, and CORBA* (13 April 1998—part of its DII COE Distributed Application Series), DISA recommends using DCE only for legacy applications.

7.1.2 .NET and Distributed Component Object Model

What Microsoft calls the .NET managed code architecture is able to transparently control application code behavior on the client and the server. .NET includes security toolset for developers to implement authentication, authorization, and cryptographic routines in their .NET-based applications, and it shifts the security decision-making role from the developer to the administrator. .NET is also said by Microsoft to eliminate many problems arising from flawed code, such as buffer overflows. Unlike DCE, CORBA, and Java RMI, .NET operates only on Microsoft Windows servers and clients, and is not supported on non-Microsoft systems.

DCOM was designed to be Microsoft's answer to DCE and CORBA. It is an object-oriented distributed computing architecture designed for use in Windows NT and 2000 applications. In essence,

DCOM is a protocol that enables software components, both Web and non-Web, to intercommunicate directly over a network. DCOM can operate across a number of network transport protocols, including HTTP for Web applications. DCOM is based on the DCE RPC specification, and uses Microsoft's COM (Microsoft's answer to the Object Request Broker) to provide its service to both ActiveX and Java applications running on Microsoft platforms. There is a DCOM tunneling transmission control protocol (TTCP) that allows DCOM to operate over TCP port 80, enabling browsers and Web servers to communicate via DCOM across proxy server or firewall boundaries.

Like all Microsoft technologies, .NET and DCOM operate only on Microsoft platforms, and cannot be used on UNIX, Linux, or any other non-Microsoft operating system. In its *Recommendations for using DCE, DCOM, and CORBA* (13 April 1998, part of its DII COE Distributed Application Series), DISA warns against using DCOM until it becomes more mature, robust, and fully understood. DISA does, however, recommend using middleware bridges into Microsoft's COM to enable intercommunication with distributed application components running on Windows systems.

7.1.3 Common Object Request Broker Architecture

In its *Recommendations for using DCE, DCOM, and CORBA* (13 April 1998, part of its DII COE Distributed Application Series), DISA recommends using CORBA for new distributed applications. It specifically prescribes the use of integrated CORBA/Internet/Java products from a single vendor unless interoperability can be verified.

According to *Recommendations for using DCE, DCOM, and CORBA*, CORBA is the middleware most widely deployed and actively used across both Windows and non-Windows systems. CORBA object request brokers (ORB) enable developers to easily distribute objects across process and machine boundaries, whereas the CORBA language (ISO/IEC DIS 14750) defines ORB interfaces to software components. The CORBA specification contains an object security model and standard security protocols. CORBA's security services include identification and authentication, privilege management, access control, message protection (confidentiality), delegation and proxy, audit, and nonrepudiation.

CORBA is integrated both with the underlying Internet protocol layers and with the Java language enables developers to combine CORBA with the existing Web infrastructure to build distributed applications more quickly and easily and to use the network-oriented features of Java to more easily integrate legacy applications into new distributed Web systems. Since 1997, the Netscape Communicator browser has included a Java-compatible version of the CORBA-compliant Borland VisiBroker. IBM's WebSphere application server incorporates a CORBA ORB, as well as Java and Enterprise JavaBean technologies.

Some developers may find use of CORBA, and specifically implementation of an ORB, too complicated for simple distributed Java applications. Those developers may wish to consider using the Java RMI middleware instead of CORBA to provide standard network and security services to their applications.

Appendix C provides links to information on CORBA and other application development middleware products.

7.1.4 Simple Object Access Protocol

SOAP is an emerging vendor-independent Web protocol specification that supports object-oriented distributed Web applications. SOAP provides a protocol for invoking for methods within servers, services, components, and objects. The SOAP specification codifies the existing practice of using XML over HTTP (or SMTP) as the mechanism for method invocation. It mandates a small number of HTTP headers that facilitate firewall and proxy filtering. The SOAP specification also mandates an XML vocabulary for representing method parameters, return values, and exceptions.

The specification of SOAP, currently at Version 1.2, is being expanded and refined by the World Wide Web Consortium (WC3). In practical terms, SOAP is comparable to the method-invocation protocols used by CORBA (IIOP), DCOM (ORPC), and Java Remote Method Invocation (RMI) (Java Remote Method Protocol, [JRMP]). SOAP differs from these other protocols, however, by being a text-based rather than binary protocol. SOAP uses XML for data encoding. That makes debugging applications based on SOAP much easier, because XML is much easier to read than is a binary data stream.

Because of its vendor-independence and dependence on truly open Web standards (XML, HTTP, SMTP), SOAP is being willingly adopted by major Web product vendors, including , a number of CORBA ORB suppliers (e.g., Iona), IBM, the Apache Software Foundation, and others. IBM, a significant contributor to the SOAP specification, has created a SOAP toolkit for Java, which it has donated to the Apache Software Foundation's XML Product. Apache, in turn, has released the open source Apache-SOAP implementation based on the toolkit. Also, Microsoft appears to be committed to using SOAP within DCOM.

Please note that the current version of this document does not discuss the security of SOAP. This exclusion stems from the fact that, although some SOAP implementations are available, the WC3's SOAP specification does not yet define the security protections for SOAP.

That said, in June 2002, VeriSign, IBM, and Microsoft submitted their jointly developed Web Services Security (WS-Security) specification to the OASIS standards body for review and, they hope, adoption. WS-Security defines a set of SOAP extensions that can be used to implement integrity and confidentiality in Web services applications based on SOAP, laying the groundwork for higher-level facilities like federation, policy, and trust. VeriSign, IBM, and Microsoft are developing five more Web security specifications that they plan to release in the next year and a half.

SOAP, along with other emerging Web technologies, such as Universal Discovery, Description, and Integration (UDDI) and Web Services Description Language (WSDL), seems positioned to become the next generation protocol for developing distributed Web applications and Web services. For this reason, DoD developers should become familiar SOAP and write their applications in a way that will not preclude the eventual adoption and integration of SOAP into DoD Web-based systems.

7.2 OTHER DEVELOPMENT TOOLS

7.2.1 Compilers and Linkers

Increase the level of type checking for C and C++ by turning on as many compiler warnings as possible, and change your code to cleanly compile with those warnings set. Strictly use ANSI prototypes in separate header (*.h*) files to ensure that all function calls use the correct types.

If using the GNU C Compiler (GCC) for C and C++ set the following (at a minimum) compilation flags: `gcc -Wall -Wpointer-arith -Wstrict-prototypes -O2 -W pedantic`.

Many C/C++ compilers can detect inaccurate format strings. For example, when using GCC, use the `__attribute__()` facility (a C extension) to mark functions that may contain inaccurate format strings. The following is an example of the language to include in the program's header (*.h*) file before compiling:

```
/* in header.h */
#ifndef __GNUC__
# define __attribute__(x) /*nothing*/
#endif

extern void logprintf(const char *format, ...)
__attribute__((format(printf,1,2)));
extern void logprintva(const char *format, va_list
args)
__attribute__((format(printf,1,0)));
```

The *format* attribute takes either `printf` or `scanf`, and the numbers that follow it are the parameter number of the format string and the first *variadic* parameter respectively. Note that there are other useful attribute facilities in GCC, such as *noreturn* and *const*.

7.2.2 Debuggers

Never deploy code compiled with debugging options on an operational system. Be aware that Windows NT and Windows 2000 have been reported to contain a critical vulnerability that enables an attacker to use operating system's standard, documented debugging interface. The interface is designed to enable the debugger to gain control of the program being tested and exploit other programs via this debugger interface, thus enabling an unprivileged user to exploit the interface to gain control of a privileged program.

Although safeguards against such exploits have long been present in UNIX and Linux (and were mentioned in Microsoft's recent Digital Rights Management patent), as of April 1, 2002, Microsoft had not released an advisory or a patch to fix this vulnerability. The person who discovered the vulnerability has posted a detailed description, including an example of an exploitation of the debugger interface. This can be downloaded from <http://www.anticracking.sk/EliCZ/bugs/DebPloit.zip>. A fix for the vulnerability

has also been published by an independent developer. This can be downloaded from the following Web page: <http://www.ntcompatible.com/article.php?sid=9313>.

7.2.3 Web Authoring Tools

Security issues with Web authoring tools arise when a tool automatically generates HTML from a visual Web page design, and that HTML includes code errors, or is so excessively large that it negatively affects client performance (due to excessively long downloads). Other problems with code produced by Web authoring tools are more nuisance problems, although at least one of these nuisances has, in fact, manifested as a kind of denial of service in that the HTML generated by the tool was so extremely optimized for a particular brand name and version of browser that the page could not be displayed at all to browsers from other vendors.

Although optimizing Web pages for a particular brand name or version of browser may seem like a reasonable practice for Web pages that will be served on an intranet that can be guaranteed to be accessed only using that brand name or version of browser, the implications for longer term extensibility and maintainability are not good. What happens if parts of the user community start using a different browser? What happens if the vendor of the browser changes its features in a later version in a way that causes the old optimized code to display incorrectly on the new browser version? What happens if you want to use a different Web authoring tool, one that may not recognize the nonstandard HTML produced by the browser-dependent tool?

It is better to write standard HTML that is not optimized for any particular browser. In this way, the Web pages you create will produce the same results regardless of what browser is used (as long as it supports the version of standard HTML you used), and their maintenance will not be dependent on use of any particular Web authoring tool.

7.2.3.1 WYSIWYG Tools versus. Text Editors and HTML Editors

Web authoring tools generally fall into one of two categories. The first is the WYSIWYG (“what you see is what you get”) tool, such as Microsoft FrontPage, Adobe Pagemill, and Macromedia Dreamweaver. WYSIWYG tools enable Web authors to create Web pages visually, similar to designing documents in a desktop publishing program, without one’s having to write a single line of HTML or XML code. The latest versions of these programs also support dynamic HTML. Although WYSIWYG page design can speed up and development process, the resulting Web page design may be limited by limitations in the tool’s ability to convert a visual design into HTML that accurately reproduces that design.

Text editors allow the developer to type as ASCII text the HTML code for the Web page being developed. Writing HTML this way is no different from the way UNIX manual authors used to create UNIX manuals: using the *vi* editor to type ASCII text with *nroff* or *troff* formatting tags (essentially the same approach used by the earliest electronic typesetting systems). The benefit of this approach is that the resulting HTML is likely to be very lean, excluding any unnecessary tags, completely understood by the developer, and easy to maintain.

HTML editors go a step beyond simple text editors. The developer still creates the Web page by coding HTML tags, but the HTML editor also provides facilities that help with coding, such as easy code navigation features, built-in code validation, interfaces for easily creating and importing style sheets including cascading style sheets (CSS), support for Web authoring languages other than HTML (e.g., ASP, PHP, VBScript, JavaScript), and libraries of canned CGI scripts.

A third approach to Web page generation is not to use an authoring tool at all, but to use the Save as HTML or Export as HTML feature built into a word processor or desktop publishing tool. The idea is that the generated HTML should look exactly like the document from which it was generated. Unfortunately, this approach to HTML generation can be quite problematic and can make page authoring more difficult and time consuming than it would be when typing the HTML line by line with a text editor.

7.2.3.1.1 Common Problems with Web Authoring Tools

The most common Web authoring tool problems fall into three categories:

1. Production of bloated code, that is, code made large by inclusion of unnecessary or irrelevant HTML tags
2. Inclusion of nonstandard tags or errors
3. Production of code that is not browser neutral.

Although these may not immediately seem like security issues, they have security implications.

7.2.3.1.1.1 Bloated Code

Web authoring tools that generate HTML, versus those that support the programmer's HTML coding (i.e., WYSIWYG tools) often insert unnecessary HTML tags, and thus unnecessarily large HTML files. Many of the default settings in Microsoft's FrontPage 2000, for example, cause the program to automatically insert irrelevant *alt* tags that mimic the file names of graphics, as well as unnecessary metadata tags. At the extreme, bloated HTML files can take an unreasonably time to download to the users' browsers, particularly if the user is connecting remotely via a dial-up link.

Always review automatically generated HTML using a text editor or HTML editor, and strip out all tags and comments that are not absolutely necessary.

7.2.3.1.1.2 Nonstandard Tags and Errors

Some Web authoring tools that automatically generate HTML may include poorly formed links and other HTML syntax errors that can cause browsers to freeze up due to the inability to parse the errored code. Errors also can create vulnerabilities associated with poorly formed URLs. An addition problem arises with tools that are optimized for only one browser version. They may generate nonstandard HTML tags (i.e., tags invented by the browser vendor but not supported by any other browsers) that

prevent the pages from being displayed correctly, or at all, on other browsers. That point is discussed in the next section.

Always review automatically generated HTML by using a text editor or HTML editor, and correct all URL references (changing relative pathnames to complete pathnames, etc.). Replace all questionable HTML tags, such as vendor proprietary tags, tags from later versions of HTML than that stated in the DOCTYPE declaration, with purely standard equivalents.

7.2.3.1.1.3 Code That Is Not Browser-Neutral

Some Web authoring tools allow developers to produce HTML that is optimized for only one brand name or version of browser. This is not a security issue itself. However, one particular tool, Microsoft's FrontPage 2000, actually defaults to generating code that is so strictly optimized for Internet Explorer that the resulting Web pages simply will not display at all in other brand names of browsers. Users of other browsers thus are shut out of the Web site.

If you use FrontPage as your Web authoring tool, be sure to turn off all of the system defaults that optimize code for Internet before generating any HTML. Better yet, use a tool that generates code that is absolutely browser neutral, producing the identical results in all browsers.

Always review automatically generated HTML using a text editor or HTML editor, and strip out all browser optimizing code. Replace all vendor proprietary tags with purely standard equivalents. Also test all Web pages for consistent presentation and behavior in both Netscape Navigator/Communicator and Microsoft Internet Explorer.

7.3 PROGRAMMING LANGUAGE SECURITY

Appendix D provides information on security issues with individual programming languages used in Web development. These languages include

- C and C++
- Visual Basic
- Java
- HTML
- XML
- ASP and JSP
- CGI and Perl
- Shell scripting languages
- TCL
- PHP
- Python.

8.0 PREPARING APPLICATIONS FOR DEPLOYMENT

8.1 PREPARING CODE FOR DEPLOYMENT

8.1.1 Remove Debugger Hooks and Other Developer Backdoors

Before deploying the application operationally, eliminate all developer backdoors and default settings from application code. Debug commands come in two distinct forms: explicit and implicit. Recognize and remove both from code before deploying it.

8.1.1.1 Explicit Debugger Commands

A name value pair left in the code or introduced as part of a URL is used to induce the server to enter debug mode. Check all URLs to ensure that they do not contain commands such as `debug=on` or `Debug=YES`. Here is an example:

```
http://www.creditunion.gov/account_check?ID=8327dsdd  
i8qjgqllkjdlas&Disp=no
```

That can be intercepted and altered by a hacker to this:

```
http://www.creditunion.gov/account_check?debug=on&ID  
=8327dsddi8qjgqllkjdlas&Disp=no
```

The attacker then observes the resultant server behavior.

Debug constructs can also be placed inside HTML code or JavaScript when a form is returned to the server, simply by adding another line element to the form construction. It would have the same results as in the URL attack as shown.

8.1.1.2 Implicit Debugger Commands

Such commands are seemingly innocuous elements placed in Web page code by the programmer to make it easier to alter the system state and speed up testing time. But if altered by a hacker, can have dramatic effects on the server. The programmer usually gives these elements obscure names, such as `fubar1` or `mycheck`, in hopes of also obscuring their purpose. For example:

```
<!-- begins -->  
<TABLE BORDER=0 ALIGN=CENTER CELLPADDING=1  
CELLSPACING=0>  
<FORM METHOD=POST  
ACTION="http://some_poll.gov/poll?1688591"  
TARGET="sometarget" FUBAR1="666">  
<INPUT TYPE=HIDDEN NAME="Poll" VALUE="1122">  
<!-- Question 1 -->
```

```
<TR>
  <TD align=left colspan=2>
  <INPUT TYPE=HIDDEN NAME="Question" VALUE="1">
  <SPAN class="Story">
```

Do a text search through your code to locate and remove all debug elements before posting the Web code, so that they cannot be found and tampered with by hackers.

Debug commands have also been known to remain in third-party applications, such as Web servers and database programs. If at all possible, scan the source code for these applications to locate and remove such commands. Or at the very least encourage the supplier of the code to do so.

8.1.2 Remove Data-Collecting Trapdoors

In the case of Web applications, the most common data-collecting trapdoor is the cookie. 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 Web sites. Nor must they be used by contractors operating Web sites on behalf of Federal agencies, unless, in addition to clear and conspicuous notice, the following conditions are met:

1. There is a compelling need to gather the data on the site.
2. Appropriate and publicly disclosed privacy safeguards for handling of information derived from cookies have been implemented.
3. The head of the agency owning the Web site has personally approved use of the data collecting cookies.

This policy was particularized for 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" (<http://www.c3i.osd.mil/org/cio/doc/cookies.html>) (13 July 2000), which states

This memorandum is to remind each Component that Department of Defense (DOD) policy prohibits the use of Web technology which collects user-identifying information such as extensive lists of previously visited sites, e-mail addresses, or other information to identify or build profiles on individual visitors to DOD publicly accessible web sites. DOD policy, however, does permit the use of "cookies" or other Web technology to collect or store non-user identifying information but only if users are advised of what information is collected or stored, why it is being done, and how it is to be used. This policy will be clarified to make clear that "persistent cookies" (i.e., those that can be used to track users over time and across different Web sites) are authorized only when there is a compelling need to gather the data on the site; appropriate technical procedures

have been established to safeguard the data; and the Secretary of Defense has personally approved use of the cookie.

Please refer to Section 5.3.1.9 for information on protecting cookies at rest and in transit.

In addition to removing any non-policy-compliant cookies that may have inadvertently been left in the application, the application code should also be carefully vetted before deployment to ensure that it does not contain any other kinds of Web bugs (see the following note) or trapdoor programs, such as malicious trapdoors, whose purpose is either to collect or tamper with data, or to open a backdoor channel over which an attacker could collect or tamper with data, on the Web server itself or on any backend servers accessed by the application.

NOTE: A Web bug is a graphic on a Web page or in an e-mail message that is designed to monitor who is reading the Web page or e-mail message. Web bugs are often invisible because they are typically only 1by1 pixel in size. They are represented as HTML IMG tags. Here is an example:

```
<IMG WIDTH=1 HEIGHT=1border=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 code should be checked to ensure that they are not being used to implement Web bugs.

8.1.3 Remove Hard-Coded Credentials

We have already established that HTML basic authentication should never be used in DoD Web applications, even over SSL-encrypted connections (see Section 3.2.6.10). Not using basic authentication should alleviate any need for hard-coded credentials in HTML pages.

NOTE: For an illustration of the how not way to implement SSO across multiple Web servers see Microsoft's workaround at <http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q281408&>. It shows a security practice that should not be duplicated in DoD Web applications.

8.1.4 Remove Default Accounts

Many COTS applications are preconfigured with at least one user (typically the administrator) activated by default; this default user generally has a standard password that can be used to compromise the system through the guessing of typical standard default passwords. Web applications often enable

multiple default accounts, including administrator, test, and guest accounts. All have widely-known default passwords.

All unnecessary accounts should be immediately disabled when the COTS Web server or other application is installed. Passwords should be changed from default on any accounts left enabled. The vendor-provided hardening scripts and vulnerability scanners should be run to ensure that no default accounts have been left active unintentionally.

8.1.5 Replace Relative Pathnames

Always use full pathnames for any URL or filename argument, for both commands and data files. Do not depend on the current directory, but explicitly set the directory in the code. Problems caused by relative pathnames include:

- Forcing the current (.) directory to be searched first
- Loading a hostile application or library routine instead of trusted software
- Complicating the search rules for DLLs, and so forth
- Introducing a vulnerability in system routines that parse filenames containing embedded spaces.

8.1.6 Remove Sensitive Comments

Do not include comments in source code that reveal potentially compromising information if that source code can possibly be viewed by others. An example is web site structure or security information. This is a particular problem with HTML source code, which can easily be viewed via the browser's View Source function. Information that absolutely should not be in HTML comments include

- Directory structures
- Location of the Web root
- Debug information
- Cookie structures
- Problems associated with development
- Developers' names, email addresses, and phone numbers.

Comments are included in HTML files in one of the following ways:

- *Structured comments*: Included regularly by members of large development teams working on large Web sites, at the top of the HTML source code page, or between the JavaScript and the remaining HTML, to let other developers know what is going on in the source code.
- *Automated comments*: Automatically added to HTML pages by many Web page generation programs. Web usage programs, these comments reveal precise information about the software

used to create the Web page (sometimes including specific release numbers). This information can be exploited by attackers to target known vulnerabilities in Web pages generated by those software programs.

- *Unstructured comments*: Informal comments inserted ad hoc by developers as memory aids. Comments 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” make a hacker’s life easier.

The following HTML comments represent security violations:

```
<!--#exec cmd="rm -rf /"-->  
<!--#include file="secretfile"-->
```

The second command appears unlikely to be a security violation, given that the *httpd* restricts the content of the file name unless the Web server has *exec* disabled.

A simple filter should be used to strip out all comments from HTML code before the pages are loaded onto the Web server. In the case of automated comments, an active filter may be required to remove comments on an ongoing basis.

8.1.7 Remove Unnecessary Files, Pathnames, and URLs

Before deploying the Web server, remove all unnecessary files. Use a staging screening process to find backup and temporary files (e.g., on UNIX, do a recursive file *grep* of all extensions that are not explicitly allowed). This will prevent file and application enumeration attacks, whereby an attacker seeks files or applications that may be exploitable or be useful in constructing an attack, such as

1. Known vulnerable third-party (COTS, open source) application files or code
2. Hidden or unreferenced files and programs unnecessarily left on the Web server by the administrator, including exploitable demo programs, sample code, installation files, and other unused programs and data files
3. Backup and temporary files that may contain sensitive information.

8.1.8 Remove Unneeded Calls

The code walk-through performed as part of the application’s testing should be used to identify and remove any calls in the application code that do not actually accomplish anything. Examples are calls to external processes or libraries that do not exist or that have been replaced.

8.2 RUN-TIME CONSIDERATIONS

8.2.1 Load Initialization Values Safely

Many programs 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, store the file in a directory other than the current directory. Also, load user defaults from a hidden file or directory in the user's home directory. If the program is *setuid/setgid*, do not read any file controlled by the user without first carefully filtering it as untrusted input. Trusted configuration values should be loaded from a different entirely (e.g., *from /etc* in UNIX).

8.3 SECURE INSTALLATION AND CONFIGURATION

8.3.1 Configure Safely and Use Safe Defaults

Faulty configuration is the source of most application security problems. When applications are installed, it is important to (1) make the initial installation secure and (2) make it easy to reconfigure the system while keeping it secure.

Do not write installation routines to install a working default password. If user accounts must be configured at installation time, assign those accounts very strong passwords, and leave it up to the administrator to reset the passwords before the system goes into production.

Configure the most restrictive access control policy possible when installing the application. It will be up to the administrator to make any changes to that policy before the application goes operational. Do not include sample working users or *allow access to all* configurations in the starting configuration for the application.

Write installation scripts to install the application software as safely as possible. By default, install all files with root read and write privileges that prevent it from being accessed by any end users.

When installing, make sure that any assumptions necessary for security are valid. For example, be sure that library routines used by the application are indeed safe on the particular platform (operating system) on which it is being installed. Also verify that the application is being installed only on the anticipated platforms before making any security assumptions about a given platform's security mechanisms and posture.

If there is a configuration language, the default should be to deny access until the user specifically grants it. Include many clear comments in the sample configuration file, if there is one, so that the administrator understands what the configuration does.

APPENDIX A: ABBREVIATIONS AND ACRONYMS

The following abbreviations and acronyms were used in this document.

ACL	Access Control List
AES	Advanced Encryption Standard
API	Application program interface
ASD C3I	Assistant Secretary of Defense for Command, Control, Communications and Intelligence
C&A	Certification and Accreditation
CAC	Common Access Card
CAPI	Cryptographic Application Programmatic Interface
CC	Common Criteria
CDR	Critical Design Review
CIO	Chief Information Officer
CERT	Computer Emergency Response Team
CGI	Common Gateway Interface
CinC	Commander in Chief
COE	Common Operating Environment
COM	Common Object Model (Microsoft)
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-The-Shelf
CRL	Certificate Revocation List
CPU	Central Processing Unit
CSP	Cryptographic Service Provider
CSS	Cascading Style Sheets
DAA	Designated Accrediting Authority
DAC	Discretionary Access Control
DAL	Data Abstraction Layer
DBMS	Database Management System
DCE	Distributed Computing Environment (OpenGroup)
DCOM	Distributed Component Object Model (Microsoft)
DES	Data Encryption Standard
DID	Defense In Depth
DII	Defense Information Infrastructure

DISA	Defense Information Systems Agency
DITSCAP	DoD Information Technology Security Certification and Accreditation Process
DHTML	Dynamic Hyper Text Markup Language
DLL	Dynamic Link Library (Microsoft)
DMC	Defense Mission Category
DMZ	Demilitarized Zone
DoS	Denial Of Service
DoD	Department of Defense
DPA	Distributed Password Authentication
DTD	Document Type Definition
EAL	Evaluation Assurance Level (Common Criteria)
E-mail	Electronic Mail
FAQ	Frequently Asked Questions
FHS	Filesystem Hierarchy Standard
FIPS	Federal Information Processing Standard (NIST)
FSO	Field Security Office
FTP	File Transfer Protocol
GCC	GNU C Compiler
GCCS	Global Command and Control System
GIF	Graphic Interchange Format
GIG	Global Information Grid
GINA	Graphical Identification and Authentication (Microsoft)
GOTS	Government-Off-The-Shelf
GSI	Grid Security Infrastructure
GSS-API	Generic Security Service Application Program Interface
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HTML	Hypertext Markup Language
I&A	Identification and Authentication
IA	Information Assurance
IASE	Information Assurance Support Environment (DISA)
IATF	Information Assurance Technical Framework (NSA)

ID	Identification, Identifier
IETF	Internet Engineering Task Force
IFS	Internal Field Separator
IP	Internet Protocol
IPSEC	IP security protocol
IIS	Internet Information Server
IPC	Interprocess Communication
ISO	International Standards Organization
IV&V	Independent Verification and Validation
ISSE	Internet System Security Engineering
J2EE	Java/Java2 Enterprise Edition (Sun Microsystems)
JAAS	Java Authentication and Authorization Service (Sun Microsystems)
JDBC	Java Database Connectivity API/service (Sun Microsystems)
JITC	Joint Interoperability Test Command
JRMI	Java Remote Method Invocation
JRMP	Java Remote Method Protocol
JSP	Java Server Pages (Sun Microsystems)
JTA	Java Transaction API (Sun Microsystems)
JVM	Java Virtual Machine (Sun Microsystems)
KRL	Key Revocation List
LDAP	Lightweight Directory Access Protocol
MAC	Mandatory Access Control; Message Authentication Check; Message Authentication Code
NFS	Network File System
NIAP	National Information Assurance Partnership (NIST)
NIST	National Institute of Standards and Technology
NSA	National Security Agency
NSS	National Security Systems
NTLM	NT LAN Manager
OCSP	Online Certificate Status Protocol
ODBC	Open Data Base Connectivity
ORB	Online Request Brokers
OS	Operating System

OSD	Office of the Secretary of Defense
OWASP	Open Web Application Security Project
PAM	Pluggable Authentication Module
PC	Personal Computer
PDF	Portable Document Format (Adobe)
PDR	Preliminary Design Review
PICS	Platform for Internet Content Selection
PKE	Public Key-Enabling
PKI	Public Key Infrastructure
PMO	Program Management Office
POP	Post Office Protocol
POSIX	Portable Operating System Interface (“X” does not stand for anything)
PRNG	Pseudorandom Number Generator
RBAC	Role-Based Access Control
RDBMS	Relational Database Management System
RFC	Request for Comments (IETF)
RMI	Remote Method Invocation (Java)
RPC	Remote Procedure Call
SBU	Sensitive But Unclassified
SDK	Software Developer’s Kit
setuid	Set User Identification (UNIX)
setgid	Set Global Identification (UNIX)
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SP	Special Publication
SPI	Service provider interface
SQL	Structured Query Language
SRR	System Requirements Review
SSE-CMM	System Security Engineering Capability Maturity Model
SSL	Secure Sockets Layer
SSO	Single Sign-On
SSPI	System Support Providers Interface (Microsoft)
ST&E	Security test and evaluation

STIG	Security Technical Implementation Guide (DISA)
TCB	Trusted Computing Base
TCP	Transfer Control Protocol
TDY	Temporary Duty Yonder
TFW	Taskforce Web (Navy)
TLS	Transport Layer Security
TOCTOU	Time Of Check/Time Of Use
TPEP	Trusted Product Evaluation Program (NSA)
TTCP	Tunneling Transmission Control Protocol
TTY, TTYP	a dumb terminal (UNIX), such as VT220 (both abbreviations derive from the word “teletype,” but no longer denote it)
UDDI	Universal Discovery, Description, and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transformation Format (E.G., UTF-8)
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WDG	Web Design Group
WSDC	Web Services Description Language
WS	Web Services
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

APPENDIX B: REFERENCES AND SUGGESTED READING

B.1 REFERENCES USED TO PREPARE THIS DOCUMENT

The following documents and on-line resources were used as sources for some of the information and concepts presented in this developer's guide.

Information Assurance Technical Framework (IATF), Version 3, September 2000

http://www.iatf.net/framework_docs/version-3_1/index.cfm

B.1.1 DESIGN AND ARCHITECTURE

Peter G. Neumann, SRI International: *CHATS (Composable High-Assurance Trustworthy Systems) Principles* (DARPA Contract No. N66001-01-C-8040 Task 1 Deliverable A003, 15 April 2002)

<http://www.csl.sri.com/users/neumann/chats2.html>

Christophe Bidan, Valérie Issarny: *Security Benefits from Software Architecture*

<http://www.inria.fr/rrrt/rr-3114.html>

Joseph Yoder, Department of Computer Science, University of Illinois at Urbana-Champaign & Jeffrey Barcalow, Reuters Information Technology: *Architectural Patterns for Enabling Application Security*

<http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/Proceedings/yoder.pdf> (PDF)

<http://www.joeyoder.com/papers/patterns/Security/appsec.doc> (Microsoft Word)

Christian Damsgaard Jensen: *Secure Software Architectures*

<http://www.ifi.uib.no/konf/nwper98/papers/jensen.ps> (PostScript)

Defense Information Systems Agency (DISA) Joint Interoperability & Engineering Organization (JIEO) Center for Computer Systems Engineering (JEXF): *Recommendations for Using DCE, DCOM, and CORBA* (MITRE-DAS-C1, 13 April 1998)

Microsoft Corporation: *Security in the Microsoft .NET Framework*

<http://www.microsoft.com/technet/itsolutions/net/evaluate/fsnetsec.asp?frame=true>

World Wide Web Consortium: *SOAP Version 1.2 Part 0: Primer* (Working Draft, 26 June 2002)

<http://www.w3.org/TR/2002/WD-soap12-part0-20020626/>

B.1.2 ACQUISITION OF THIRD-PARTY PRODUCTS

National Security Telecommunications Information Systems Security Policy 11, "National Policy Governing the Acquisition of Information Assurance and IA-Enabled Information Technology (IT) Products" (NSTISSP 11)

<http://web1.deskbook.osd.mil/reflib/MFED/006MV/006MVDOC.HTM>
http://www.nstissc.gov/Assets/pdf/nstissp_11.pdf

B.1.3 IMPLEMENTATION

David A. Wheeler: *Secure Programming for Linux and UNIX HOWTO*

<http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html>

Joanna Oja, Department of Computer Science, University of Helsinki: *Secure Software Development* (white paper and slide presentation)

<http://www.tml.hut.fi/~viu/distsec/secure-software-dev.pdf> (white paper)

<http://www.tml.hut.fi/~viu/distsec/secure-software.pdf> (slide presentation)

Razvan Peteanu: *Best Practices for Secure Development*

http://members.rogers.com/razvan.peteanu/best_prac_for_sec_dev4.pdf

Razvan Peteanu: *Best Practices for Secure Web Development*

http://www.cert.pl/PDF/secure_webdev-3.0.pdf

<http://www ldc.lu.se/security/BestPracticeWWW.pdf> (mirror)

Steve Pettit, Sanctum Inc.: *The Anatomy of a Web Application – Security Considerations*

<http://www.securewebonline.com/AnatomyApplicationFINAL.pdf>

http://www.isacantx.org/docs/Web_Server.pdf (mirror)

Best Practices for Secure Web Development

http://softwaredev.earthweb.com/security/article/0,,10527_640861,00.html (part 1)

http://softwaredev.earthweb.com/security/article/0,,12013_640891,00.html (part 2)

Open Web Application Security Project (OWASP): *A Guide to Building Secure Web Applications and Web Services*

<http://www.cgisecurity.com/owasp/OWASPBuidingSecureWebApplicationsAndWebServices-V1.0.pdf>

National Aeronautics and Space Administration Dryden Flight Research Center: *Software Assurance*

<http://www.dfrc.nasa.gov/DMS/pdf/DCP-S-007.pdf>

B.1.3.1 Cryptographic Implementation

Application Security, Inc.: *Encryption of Data at Rest*

http://www.appsecinc.com/presentations/Encryption_of_Data_at_Rest.pdf

<http://hoohoo.ncsa.uiuc.edu/cgi/security.html>

Cambridge University: *PKCS#11 Interface*

<http://smartcard.caret.cam.ac.uk/pkcs11.html>

B.1.3.2 Avoiding Malicious Content

Computer Emergency Response Team Coordination Center: *Understanding Malicious Content Mitigation for Web Developers*

http://www.cert.org/tech_tips/malicious_code_mitigation.html

B.1.3.3 Mobile Code

DoD Chief Information Officer: *Policy Guidance for use of Mobile Code Technologies in Department of Defense (DoD) Information Systems* (7 November 2000)

<http://www.c3i.osd.mil/org/cio/doc/mobile-code11-7-00.html> (HTML)

<http://www.c3i.osd.mil/org/cio/doc/mobile-code11-7-00.pdf> (PDF)

<http://www.dfas.mil/technology/pal/security/asecroot/Artifacts/ShortVer/Mobile.doc> (Microsoft Word)

B.1.3.4 Security Issues of Specific Programming Languages

Eduardo B. Fernandez: *Computer Data Security*, Chapter 11, Application and Language Security

<http://www.cse.fau.edu/~ed/Microsoft%20Word%20-%20AppSec.pdf> (PDF)

<http://www.cse.fau.edu/~ed/AppSec.doc> (Microsoft Word)

Sun Microsystems, Inc.: *Security Code Guidelines*

<http://java.sun.com/security/seccodeguide.html>

David A. Wheeler: *Java Security*

<http://dwheeler.com/javasec/javasec.pdf>

Gary McGraw and Edward Felten: “Twelve Rules for Developing More Secure Java Code” (*JavaWorld*, December 1998)

<http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html>

National Center for Supercomputing Applications: *Writing Secure CGI Scripts*

<http://hoohoo.ncsa.uiuc.edu/cgi/security.html>

B.1.3.5 Testing

Naval Research Laboratory: *Handbook for the Computer*

Security Certification of Trusted Systems, Chapter 10, Penetration Testing (NRL Technical Memorandum 5540:082A, 24 January 1995)

<http://chacs.nrl.navy.mil/publications/handbook/PENET.pdf>

Halvar Flake: *Auditing Closed-Source Applications*

<http://www.blackhat.com/presentations/bh-europe-00/HalvarFlake/HalvarFlake.ppt>

B.2 SUGGESTED FURTHER READING

The following documents and on-line resources were also found to be generally helpful, and encourage developers to refer to them for additional information, examples, and approaches to implementing security in Web applications.

Lincoln D. Stein and John N. Stewart: *The World Wide Web Security FAQ*

<http://www.w3.org/Security/faq/www-security-faq.html>

Web Application Security FAQ

http://www.securewebonline.com/app_faq.htm

Debbie Carson: *Web Application Security*

<http://java.sun.com/webservices/docs/ea2/tutorial/doc/WebAppSecurity.html>

Matthew Levine, @stake: *The Importance of Application Security* (March 2002)

http://www.atstake.com/research/reports/atstake_application_security.pdf

Jeremiah Grossman, WhiteHat Security, Inc.: *Web Application Security: The Land That Information Security Forgot*

<http://www.blackhat.com/presentations/bh-europe-01/jeremiah-grossman/grossman.ppt>

http://www.whitehatsec.com/blackhat_amsterdam2001/BlackHat_Europe2001_Presentation.ppt (mirror)

Security Focus Online Web Application Security Forum

<http://online.securityfocus.com/archive/107>

B.2.1 DESIGN AND ARCHITECTURE

P. Boudra, Jr., Office of Infosec Systems Engineering (I9), Information Systems Security Organization, National Security Agency: *Rules of System Composition: Principles of Secure System Design* (Technical Report 1-93, Library No. S-240, 330, March 1993)

Eduardo B. Fernandez: *Computer Data Security*, Chapter 5, The Design of Secure Systems

<http://www.cse.fau.edu/~ed/SecSysDesign.doc>

Microsoft Corporation: *Designing Secure Applications*

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mwSDK/html/mwcondesigningsecureapplications.asp>

Premkumar T. Devanbu, Philip W-L Fong, and Stuart G. Stubblebine: *Techniques for Trusted Software Engineering*

http://uuu.sourceforge.net/docs/Trusted_Software_Engineering.pdf

Sotiris Ioannidis and Steven M. Bellovin: *Sub-Operating Systems: A New Approach to Application Security*

<http://www.research.att.com/~smb/papers/subos.pdf>

Paul A. Green, Jr.: *The Art of Creating Reliable Software-Based Systems Using Off-the-Shelf Software Components*

http://ftp.stratus.com/pub/vos/doc/papers/SRDS_Paper_1997.doc

B.2.1.1 Security Middleware Frameworks

Chris Cleeland, Principal Software Engineer and Rob Martin, Senior Software Engineer, Object Computing, Inc. (OCI): *CORBA Security: An Overview*

<http://www.ociweb.com/cnb/CORBANewsBrief-200205.html>

Microsoft Security in the .NET Framework

<http://msdn.microsoft.com/netframework/techinfo/articles/security/default.asp>

About .NET Security

http://www.gotdotnet.com/team/clr/about_security.aspx

B.2.2 IMPLEMENTATION

National Center for Supercomputing Applications (NCSA): *Secure Programming Guidelines*

<http://archive.ncsa.uiuc.edu/General/Grid/ACES/security/programming/>

Victor A. Rodriguez: *The Secure Programming Standards Methodology Manual*

<http://uk.osstmm.org/spsmm.htm>

Secure Programming, v1.00

<http://www.cli.di.unipi.it/~zoppi/docs/secprog.html>

Frédéric Raynal, Christophe Blaess and Christophe Grenier: *Avoiding Security Holes When Developing an Application—Part 1*

<http://mercury.chem.pitt.edu/~tiho/LinuxFocus/English/January2001/article182.shtml>

Steve Bellovin: *Shifting the Odds—Writing (More) Secure Software*
<http://www.research.att.com/~smb/talks/odds.pdf>

B.2.2.1 UNIX Applications

Thamer Al-Herbish: *Secure UNIX Programming FAQ*
<http://www.whitefang.com/sup/>

Australian CERT: *Practical UNIX and Internet Security—Secure Programming Checklist*
ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist

B.2.2.2 Implementing Specific Security Mechanisms

B.2.2.2.1 Identification and Authentication

Nick Kew: *Login on the Web*
<http://www.webthing.com/tutorials/login.html>

Internet Engineering Task Force S/Key-related Requests for Comments and Charters:
<http://www.ietf.org/html.charters/otp-charter.html>

Dion Almaer: “Web Form Based Authentication” (*O’Reilly OnJava.com*, 8 June 2001)
<http://www.onjava.com/pub/a/onjava/2001/08/06/webform.html>

B.2.2.2.2 Public Key Infrastructure and Public Key-Environment

DISA Information Assurance Support Environment Document Library
<https://iase.disa.mil/documentlib.html#PKIDOCS>

DoD PK-Enabling Web site
<http://www.dodpke.com/sitemap.asp>

Specifically, see the following documents:

- Department of Defense (DoD) Class 3 Public Key Infrastructure (PKI) Criteria for Selecting Cryptographic Toolkits (Draft, 20 November 2000)
- DoD CLASS 3 PKI Public Key-Enabling of Applications (29 September 2000)
- DoD CLASS 3 PKI Public Key-Enabled Application Requirements, Version 1.0 (13 July 2000)

B.2.2.2.3 Symmetric Cryptography

NSA Advanced Encryption Standard Algorithm Validation List

<http://csrc.nist.gov/cryptval/aes/aesval.html>

NSA Triple Data Encryption Standard Validation List

<http://csrc.nist.gov/cryptval/des/tripledesval.html>

National Institute of Standards and Technology (NIST) Cryptographic Toolkit

<http://csrc.nist.gov/encryption/>

Microsoft Enhanced Cryptographic Service Provider (NSA approved for 3-DES)

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/security/security/microsoft_enhanced_cryptographic_provider.asp

B.2.2.2.4 Integrity Controls

F. Chang, A. Itzkovitz, and V. Karamcheti: *Secure, User-Level Resource-Constrained Sandboxing*

http://csdocs.cs.nyu.edu/Dienst/UI/2.0/Describe/ncstrl.nyu_cs%2fTR1999-795

Anurag Acharya and Mandar Raje: *MAPbox—Using Parameterized Behavior Classes to Confine Applications*

<http://citeseer.nj.nec.com/cache/papers/cs/8626/http:zSzzSzwww.cs.ucsb.edu:zSszTRszSztechreportszSszTRCS99-15.pdf/acharya00mapbox.pdf>

Wayne Schroeder, San Diego Supercomputer Center, University of California at San Diego: *The SDSC Encryption/Authentication (SEA) System* (April 1998)

http://www.sdsc.edu/~schroede/sea_cpe.html

B.2.2.3 Understanding and Minimizing Specific Vulnerabilities

B.2.2.3.1 Mobile Code Use

Dan Seth Wallach: *A New Approach to Mobile Code Security* (January 1999)

<http://www.cs.princeton.edu/sip/pub/dwallach-dissertation.php3>

Richard Drews Dean: *Formal Aspects of Mobile Code Security* (January 1999)

<http://www.cs.princeton.edu/sip/pub/ddean-dissertation.php3>

Philip W. L. Fong and Robert D. Cameron: *Proof Linking—An Architecture for Modular Verification of Dynamically-Linked Mobile Code*

<http://www.cs.sfu.ca/people/GradStudents/pwfong/personal/Pub/fse98.pdf>

B.2.2.3.2 Buffer Overflow

Susan Gerhart, Embry-Riddle Aeronautical University: *How Do Buffer Overflow Attacks Work?*
<http://nsfsecurity.pr.erau.edu/bom/index.html>

Jedidiah R. Crandall, Susan L. Gerhart and Jan G. Hogle, Embry-Riddle Aeronautical University: *Driving Home the Buffer Overflow Problem: A Training Module for Programmers and Managers*
<http://nsfsecurity.pr.erau.edu/Papers/ncisse2002.pdf> (PDF)
<http://nsfsecurity.pr.erau.edu/Talks/ncisse2002.ppt> (PowerPoint)

SecuriTeam: *Writing Buffer Overflow Exploits—A Tutorial for Beginners*
<http://www.securiteam.com/securityreviews/5OP0B006UQ.html>

Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, Oregon Graduate School Institute of Science and Technology: *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*
<http://immunix.org/StackGuard/discex00.pdf>
<http://suif.stanford.edu/~courses/cs343/writing/buffer-ovf.pdf> (mirror)

Crispin Cowan, Calton Pu, David Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, Oregon Graduate School Institute of Science and Technology, and Heather Hinton, Ryerson Polytechnic Institute: *StackGuard: Automatic Adaptive Prevention of Buffer-Overflow Attacks*
<http://www.immunix.org/StackGuard/usenixsc98.pdf>

Gary McGraw and John Viega: *Make Your Software Behave: Learning the Basics of Buffer Overflow*
<http://www-106.ibm.com/developerworks/library/overflows/>

Gary McGraw and John Viega: *Make Your Software Behave: Preventing Buffer Overflows*
<http://www-106.ibm.com/developerworks/library/buffer-defend.html>

Pierre-Alain Fayolle and Vincent Glaume, ENSEIRB Networks and Distributed Systems: *A Buffer Overflow Study: Attacks and Defenses*
<http://www.enseirb.fr/~glaume/bof/report.html>

David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, University of California, Berkeley: *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*
http://www.simovits.com/archive/ndss_039.pdf (white paper)
<http://web.yl.is.s.u-tokyo.ac.jp/meeting/amo/doc/amo-0516.ppt> (PowerPoint presentation)

The Tao of Windows Buffer Overflow
http://www.cultdeadcow.com/cDc_files/cDc-351/

Smashing the Stack for Fun and Profit

<http://downloads.securityfocus.com/library/P49-14.txt>

Smashing the Stack

<http://destroy.net/machines/security/>

B.2.2.3.3 Format String Attacks

Tim Newsham, Guardent, Inc.: *Format String Attacks*

<http://www.guardent.com/docs/FormatString.PDF>

TESO Security Group: *Exploiting Format String Vulnerabilities*, Version 1.0

<http://www.team-teso.net/releases/formatstring.pdf>

B.2.2.3.4 Structured Query Language Injection

SPI Dynamics: *SQL Injection: Are Your Web Applications Vulnerable?*

<http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>

SQL Injection FAQ

<http://www.sqlsecurity.com/faq-inj.asp>

SecuriTeam: *SQL Injection Walkthrough*

<http://www.securiteam.com/securityreviews/5DP0N1P76E.html>

SecuriTeam: *CRLF Injection*

<http://www.securiteam.com/securityreviews/5WP022K75O.html>

B.2.2.4 Security Issues of Specific Programming Languages

B.2.2.4.1 Java

OpenGroupX/Open Single Sign-On Service Pluggable Authentication Modules (XXSO-PAM) Specification

<http://www.opengroup.org/pubs/catalog/p702.htm>

Sun Microsystems: Java Cryptography Extension (JCE)

<http://java.sun.com/products/jce/index.html>

Sun Microsystems: Java Secure Socket Extension (JSSE)

<http://java.sun.com/products/jsse/>

Java Community Process, Java Specification Request: Java Certification Path Application Program Interface (API)

<http://jcp.org/jsr/detail/055.jsp>

Sun Microsystems, Inc.: Pluggable Authentication Modules in Solaris and Java

<http://www.sun.com/software/solaris/pam/>

Sun Microsystems: Java Authentication and Authorization Service (JAAS)

<http://java.sun.com/security/jaas/doc/pam.html>

Sun Microsystems: *Java Security Guides*

<http://java.sun.com/products/jdk/1.2/docs/guide/security/>

The Java Security Website

<http://www.cigital.com/javasecurity/>

Java Security FAQ Home Page

<http://www.jguru.com/faq/Security>

Todd Sundsted: *Secure Your Java Apps from End to End*, Part –The Foundation of Java Security: Virtual Machine and Byte Code Security

<http://www.javaworld.com/javaworld/jw-06-2001/jw-0615-howto.html>

Todd Sundsted: *Secure Your Java Apps from End to End*, Part 2–Don't Let Flaws Compromise Application Security

<http://www.javaworld.com/javaworld/jw-07-2001/jw-0713-howto.html>

The Java Tutorial: *Essential Java Classes–Handling Errors with Exceptions*

<http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html>

The Java Tutorial: *Security in Java 2 SDK 1.2*

<http://java.sun.com/docs/books/tutorial/security1.2/index.html>

The Java Tutorial: *Security in JDK 1.1*

<http://java.sun.com/docs/books/tutorial/security1.1/index.html>

Li Gong's Java Security Homepage

<http://java.sun.com/people/gong/java/security.html>

Gary McGraw and Edward Felten: "Twelve Rules for Developing More Secure Java Code"
(*JavaWorld*, December 1998)

<http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html>

Raghavan N. Srinivas: “Java Security Evolution and Concepts” (*JavaWorld*, July 2000)

<http://www.javaworld.com/javaworld/jw-07-2000/jw-0728-security.html>

Lujo Bauer, Andrew W. Appel, and Edward W. Felten: *Mechanisms for Secure Modular Programming in Java* (Princeton University Computer Science Technical Report TR-603-99, July 1999)

<http://ncstrl.cs.princeton.edu/expand.php3?id=TR-603-99>

Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz: *Java Security: Web Browsers and Beyond* (Princeton University Computer Science Technical Report TR-566-97, February 1997)

<http://ncstrl.cs.princeton.edu/expand.php3?id=TR-566-97>

Mikael Fiil: Demo of a CA Certified and DSA Signed Applet for the Sun Java Plugin

<http://www.netbizz.dk/koncepter/JavaSign/description.html>

Dirk Balfanz and Ed Felten: *A Java Filter* (Princeton University Department of Computer Science Technical Report 567-97, September 1997)

<http://www.cs.princeton.edu/sip/pub/javafilter.php3>

B.2.2.4.2 Common Gateway Interface and Perl

Michael Van Biesbrouck: *Tutorial on Writing Secure CGI Scripts*

<http://www.csclub.uwaterloo.ca/u/mlvanbie/cgisec/>

John Halperin, Maurice L. Marvin, Dave Andersen, Zygo Blaxell, Joe Sparrow, Keith Golden, James W. Abendschan, Jennifer Myers, Jarle Fredrik Greipsland, and David Sacerdote: *Safe CGI Programming*

<http://www.improving.org/paulp/cgi-security/safe-cgi.txt>

Perl 5.6 Documentation: *perlse – Perl Security*

<http://www.perldoc.com/perl5.6/pod/perlsec.html>

“Perl CGI Problems” (*Phrack Magazine*, Volume 9, Issue 55, 9 September 1999)

<http://www.insecure.org/news/P55-07.txt>

CGI/Perl Taint Mode FAQ

<http://www.gunther.web66.com/FAQS/taintmode.html>

Abhijit Menon-Sen: *Symmetric Cryptography in Perl*

<http://www.perl.com/pub/a/2001/07/10/crypto.html>

B.2.2.4.3 Extensible Markup Language and Simple Object Access Protocol

Pete Lindstrom: "Special Report: The Language of XML Security" (*Network Magazine*, 5 June 2001)

<http://www.networkmagazine.com/article/NMG20010518S0010>

XMLsec, Inc.: *Why XML Security?*

<http://www.xmlsec.com/WhyXMLSecurity.html>

XML Security Page

http://www.nue.et-inf.uni-siegen.de/~geuer-pollmann/xml_security.html

XML Digital Signature (xmldsig) RFC

<http://www.ietf.org/html.charters/xmldsig-charter.html>

<http://www.w3.org/Signature/>

Murdoch Mactaggart: *Enabling XML Security—An Introduction to XML Encryption and XML Signature*

<http://www-106.ibm.com/developerworks/security/library/s-xmlsec.html?dwzone=security>

XML Access Control Language

<http://xml.coverpages.org/xacl.html>

Microsoft Corporation: *XML Web Services Security*

<http://msdn.microsoft.com/vstudio/techinfo/articles/XMLwebservices/security.asp>

Ernesto Damiani, Università di Milano; Sabrina De Capitani di Vimercati, Università di Brescia; Stefano Paraboschi, Politecnico di Milano; and Pierangela Samarati, Università di Milano: *Design and Implementation of an Access Control Processor for XML Documents*

<http://www9.org/w9cdrom/419/419.html>

Kirill Gavrylyuk, Microsoft Corporation: *Building Secure Web Services with Microsoft SOAP Toolkit 2.0*

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsrvspec/html/soapsecurity.asp>

IBM's Proposed Web Services Security (WS-Security) Enhancements to SOAP

<http://www-106.ibm.com/developerworks/library/ws-secure/>

B.2.2.4.4 C and C++

SecuriTeam: *Using Environment for Returning into Lib C*

<http://www.securiteam.com/securityreviews/5HP020A6MG.html>

B.2.2.4.5 Avoiding Vulnerabilities Associated with Shellcode

Chris Anley, Next Generation Security Software: *Creating Arbitrary Shellcode in Unicode Expanded Strings—The “Venetian” Exploit*

<http://www.nextgenss.com/papers/unicodebo.pdf>

Next Generation Security Technologies: *Polymorphic Shellcodes vs. Application IDSs*

http://www.ngsec.com/docs/whitepapers/polymorphic_shellcodes_vs_app_IDSs.PDF

Frédéric Raynal, Christophe Blaess, and Christophe Grenier: *Avoiding Security Holes When Developing an Application – Part 2: Memory, Stack and Functions, Shellcode*

<http://www.linuxfocus.org/English/March2001/article183.meta.shtml>

B.2.2.4.6 PHP

Shaun Clowes, SecureReality.com: “A Study In Scarlet—Exploiting Common Vulnerabilities in PHP (*LinuxSecurity News* 6 July 2001)

http://www.linuxsecurity.com/articles/documentation_article-3290.html

B.2.2.5 Application Platform Security: Web Servers

B.2.2.5.1 General Web Server Security

NSA: *Web Server Protection Profile*, Version 0.6, 31 July 2001 (draft)

http://www.iatf.net/protection_profiles/file_serve.cfm?chapter=web_server_draft_07_31_01.pdf

B.2.2.5.2 Apache Web Server Security

Cross-Site Scripting Info

<http://httpd.apache.org/info/css-security/>

B.2.2.5.3 Java Web Server Security

Removing Examples and Unnecessary Servlets

<http://www.sun.com/software/jwebserver/faq/jwsca-2000-02.html>

B.2.2.5.4 Lotus Domino Web Server Security

David Litchfield, Next Generation Security Software: *Hackproofing Lotus Domino Web Server*

<http://www.nextgenss.com/papers/hpldws.pdf>

B.2.2.5.5 Microsoft Internet Information Server Web Server Security

Microsoft Corporation: *HOWTO Review ASP Code for CSSI (Cross-Site Scripting Security Issues) Vulnerability*

<http://support.microsoft.com/default.aspx?scid=kb:EN-US;q253119>

Microsoft Corporation: *HOWTO Review MTS/ASP Code for CSSI Vulnerability*

<http://support.microsoft.com/default.aspx?scid=kb:EN-US;q253121>

Microsoft Corporation: *HOWTO Review Visual InterDev Generated Code for CSSI Vulnerability*

<http://support.microsoft.com/default.aspx?scid=kb:EN-US;q253120>

B.2.2.5.6 iPlanet Portal Server Security

iPlanet Portal Server Pluggable Authentication API

<http://docs.iplanet.com/docs/manuals/portal/30/progref/pluggabl.htm>

B.2.2.6 Database Application Security

B.2.2.6.1 Oracle Applications

William T. Abram: *Developing a Secure Oracle Database Application* (GSEC v1.2f, 27 December 2001)

http://www.giac.org/practical/William_Abrams_GSEC.doc

David Litchfield, Next Generation Security Software: *Hackproofing Oracle Application Server: A Guide to Securing Oracle 9*

<http://www.nextgenss.com/papers/hpoas.pdf>

Aaron Newman, Application Security Inc.: *Protecting Oracle Databases* (and slide presentation)

http://www.appsecinc.com/presentations/Protecting_Oracle_Databases_White_Paper.pdf
(white paper)

http://www.appsecinc.com/presentations/oracle_security.ppt (PowerPoint presentation)

Next Generation Security Software: *Oracle Application Server Buffer Overflow*

<http://www.ngssoftware.com/papers/bufferoverflowpaper.rtf>

Oracle Corporation: *Oracle Application Server Security Issue—Problem Description and Immediate Methods for Eliminating the Problem*

<http://technet.oracle.com/products/oas/index2.htm?Support&pdf/owsbina.pdf>

Chris Anley, Next Generation Security Software: *Advanced SQL Injection in SQL Server Applications*

http://www.nextgenss.com/papers/advanced_sql_injection.pdf

B.2.2.6.2 Microsoft SQL Server Applications

SPI Dynamics: *SQL Injection: Are Your Web Applications Vulnerable?*

<http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>

David Litchfield, Next Generation Security Software: *Web Application Disassembly with ODBC Error Messages*

<http://www.nextgenss.com/papers/webappdis.doc>

B.2.2.7 Browser Security

National Security Telecommunications and Information Systems Security Committee (NSTISSC) Secretariat (I42), National Security Agency (NSA): *Advisory Memorandum on Web Browser Security Vulnerabilities* (NSTISSAM INFOSEC 3-00, August 2000)

<http://csrc.nist.gov/publications/secpubs/web-secvul.pdf>

http://www.nstissc.gov/Assets/pdf/NSTISSAM_INFOSEC_3-00.pdf (mirror)

NSA: *Web Browser Protection Profile*, Version 0.5, 20 (Draft April 2001)

http://www.iaf.net/protection_profiles/file_serve.cfm?chapter=web_browser_pp.pdf

B.2.3 TESTING

David A. Wagner: *Static Analysis and Computer Security—New Techniques for Software Assurance*

<http://www.cs.berkeley.edu/~daw/papers/phd-dis.ps>

Adam Shostack: *Security Code Review Guidelines*

<http://www.homeport.org/~adam/review.html>

Eric Thomas Black: *Basic Web Application Vulnerability Tests*

<http://www.well.com/user/limbo/websecurity.html>

B.2.4 RETROFITTING SECURITY INTO THIRD-PARTY AND LEGACY APPLICATIONS

Crispin Cowan and Calton Pu, Oregon Graduate Institute of Science and Technology: *Survivability from a Sow's Ear: The Retrofit Security Requirement*

<http://www.cse.ogi.edu/DISC/projects/immunix/isw98.pdf>

Timothy Fraser, Lee Badger, and Mark Feldman, TIS Labs, Network Associates, Inc.: *Hardening COTS Software with Generic Software Wrappers*

<ftp://ftp.tislabs.com/pub/wrappers/papers/wrap-oak99.ps> (PostScript)

Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole, WireX Communications, Inc.: *The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques*

<http://www.immunix.org/autonomix/crackerpatch.pdf>

Crispin Cowan, WireX Communications, Inc.: “*Immunixing*” *Open Source*

<http://www.schafercorp-ballston.com/chatsworkshop1/cowan.pdf>

John C. Dean, National Research Council Canada Software Engineering Group; and Li Li, Entrust Technologies: *Issues in Developing Security Wrapper Technology for COTS Software Products*

<http://link.springer.de/link/service/series/0558/bibs/2255/22550076.htm>

Jamie Payton, Gerur Jónsdóttir, Daniel Flagg, and Rose Gamble, Software Engineering and Architecture Team, Department of Mathematical and Computer Sciences, University of Tulsa: *Merging Integration Solutions for Architecture and Security Mismatch*

<http://link.springer.de/link/service/series/0558/bibs/2255/22550199.htm>

Peter Sewell: *Secure Composition of Untrusted Code—Wrappers and Causality Types*

<http://www.cl.cam.ac.uk/users/pes20/wrapypes-tr.ps> (PostScript)

B.3 BOOKS

The following books may also be helpful to Web application developers. The uniform resource locators (URLs) link to information online about these books. This list does not purport to be exhaustive.

Joel Scambray and Mike Shema: *Hacking Exposed, Web Applications*

<http://www.mcgraw-hill.co.uk/html/007222438X.html>

(NOTE: a McGraw Hill UK URL is given because the U.S. site is broken and will not return information about the book.)

Ulrich Lang and Rudolf Schreiner: *Developing Secure Distributed Systems with CORBA*

<http://www.objectsecurity.com/book.html>

Michael Howard and David LeBlanc, *Writing Secure Code*

<http://www.microsoft.com/mspress/books/5612.asp>

John Viega: *Building Secure Software*

<http://www.buildingsecuresoftware.com/>

John Viega and Gary McGraw: *Building Secure Software: How to Avoid Security Problems the Right Way*

http://www.pearsonptg.com/book_detail/0,3771,020172152X,00.html

Gary McGraw and Ed Felten: *Securing Java* (1999)

<http://www.securingjava.com/>

Li Gong: *Inside Java 2 Platform Security* (1999)

<http://java.sun.com/people/gong/java/awlbook1.html>

Scott Oaks: *Java Security*

<http://www.oreilly.com/catalog/javasec2/>

Rich Helton and Johnnie Helton: *Java Application Security Architecture*

<http://www.wiley.com/cda/product/0,,0764549286,00.html>

Rahul Sharma, Beth Stearns and Tony Ng: *J2EE Connector Architecture and Enterprise Application Integration*

<http://java.sun.com/docs/books/j2eeconnector/>

Michael Howard: *Designing Secure Web-Based Applications for Microsoft Windows 2000*

<http://www.microsoft.com/mspress/books/4293.asp>

APPENDIX C: THIRD-PARTY SECURITY TOOLS

C.1 SELECTING THIRD-PARTY TOOLS

C.1.1 CATEGORIES OF TOOLS

The third-party tools for application developers listed in this appendix fall into three general categories from a procurement standpoint: (1) government off-the-shelf (GOTS), (2) commercial off-the-shelf (COTS), and (3) public domain. There are various subcategories within some of these categories.

C.1.1.1 Government off-the-shelf

GOTS indicates that Government funding paid for development of the tool. The developer may be either a government organization or, more likely, a contractor and possibly a commercial vendor directly contracted and paid by the government to develop the tool.

C.1.1.2 Commercial Off-the-shelf

COTS tools are developed commercially with no government funding. Any price discount to government users must be negotiated contractually.

C.1.1.3 Public Domain

Department of Defense Directive 8500aa, “Information Assurance (IA),” paragraph 4.16 states: “Public domain software products shall only be used in DoD information systems with a validated requirement, an assessment of the product’s risk, and an approval for use by the responsible Designated Approving Authority (DAA).”

That said, as you will observe from the tables that follow, there are a significant number of useful open source tools that could be very helpful to developers of DoD Web applications. We strongly encourage you to research these tools and to make the case for their use in your Web applications. It takes only one approval of a public domain tool to establish a precedent for allowing its use.

The public domain tools of interest to application developers are, by and large, open source, versus shareware or freeware. With the advent of the open source license, there has been some blurring of the distinctions between these three types of public domain software distribution. Still, a description of each type of public domain license has been included here for information purposes.

C.1.1.3.1 Open Source

Open source software is distributed over the Internet, often in source code form, under an open source distribution license. The license typically allows for free use of the software unless it is incorporated and resold in a commercial product, in which case a royalty must be paid to the developer or supplier for

each derivative license sold. Details of different individual and organizational open source licenses vary, and the license for a particular tool should be reviewed to ensure that the tool may indeed be used free of charge.

C.1.1.3.2 Shareware

Shareware is distributed over the Internet in either executable form or source code form, under a shareware license that typically requires payment of a nominal fee to the developer or supplier to cover his her costs of development, distribution, and support. Shareware licenses prohibit the customer from repackaging and reselling the software for a profit. Shareware that is repackaged for sale must be licensed with appropriate royalties paid to the original developer supplier.

C.1.1.3.3 Freeware

Freeware is distributed over the Internet in either executable form or source code form. It is free of charge with the understanding that the software will not be repackaged and sold. Freeware that is repackaged for sale must be licensed with appropriate royalties paid to the original developer or supplier.

C.1.1.4 Selection Criteria

Assessing and selecting the right third-party security tool or technology will require a significant amount of research and analysis. DoD information technology (IT) procurement policy applies to the entire range of possible IT products, and thus it may be difficult to derive guidance from it that is specifically targeted to consideration any particular technology. Similarly, technology evaluation criteria such as the Common Criteria (validated under the National Information Assurance Partnership [NIAP]), National Institute of Standards and Technology (NIST) technical certification criteria, and others focus predominantly on one aspect of a technology. Examples are its security assurance and its compliance with standards. Collectively, however, these various policies and criteria may be used as the source for deriving a set of relevant assessment criteria for a particular type of technology.

In this section, we have attempted to provide a kind of derived set of assessment criteria to be used when evaluating third-party security tools for use in the development of DoD Web applications. These are informal criteria, intended to provide a set of considerations to keep in mind when researching and assessing tools that you may then want to formally procure following standard DoD procedures and policy.

C.1.1.4.1 NSTISSP 11 and the Need for NIAP Certification of COTS IA Products

Regardless of the other selection criteria established for COTS product procurements, National Security Telecommunications Information Systems Security Policy (NSTISSP) 11, National Policy Governing the Acquisition of Information Assurance and IA-Enabled Information Technology (IT) Products states that, as of July 2001, all IA and IA-enabled products procured for inclusion in DoD systems that will be used to enter, process, store, display, or transmit national security information must

be NIAP validated (see <http://www.eescom.com/nstissp11.pdf>). Under the same policy, GOTS products must be evaluated or approved not by NIAP, but by the National Security Agency (NSA). *Note: The policy also includes a provision for obtaining waivers to these requirements. Until a wide range of NIAP-certified IA and IA-enabled products are available that cover the full spectrum of DoD application functional requirements, that provision may be the only viable approach to complying with NSTISSP 11.*

C.1.1.4.2 Functional and Quality Criteria

- *Functionality and performance* — The tool supports the functional and performance requirements for the capability it is intended to provide.
- *Correctness* — The tool operates according to its specifications and does not contain undocumented features, errors and bugs, or security vulnerabilities.
- *Security* — The tool will not introduce security vulnerabilities into any of the other Web application components or interfaces between those components.
- *Integration* — The tool includes secure (non compromising) application program interfaces (APIs) for integration into the Web application; whenever possible, these APIs are based on published standards.
- *Auditability* — The tool's security events can be defined to the Web application's audit system and monitored by its violation detection system.
- *Performance impact* — The performance overhead added by the tool is not unacceptable
- *User friendliness* — If the tool has a user interface, it is not difficult to use and does not require a significant amount of training.
- *Product and supplier track record* — The tool itself has a reputation for quality and reliability. The tool's supplier is known to be reliable and is not likely to go out of business during the time the tool will be used in your application. If your organization has had no previous dealings with a particular tool or its supplier, you should ask the supplier for a list of customer references. You should also post some messages on developers' forums (newsgroups, bulletin boards, etc.), asking for other developers to share their experiences with the supplier and, specifically, the technology you are considering.

C.1.1.4.3 Independent Security Certification

In addition to satisfying the functional criteria discussed, tools that perform security functions (i.e., trusted functions, privileged functions, sensitive functions) should ideally have had their security functionality and assurance independently certified. Indeed, there is a DoD policy for security technologies that states they must be evaluated against the CC under NIAP.

Unfortunately, it is likely that this policy will take a long time to be fully implemented, if it ever can be. The time and cost for vendors to attain NIAP CC evaluations for their products will probably limit the number of evaluated products available at any given time. In many cases, the types of COTS tools you will find useful are not unlikely to be submitted for evaluation; nor are public domain suppliers likely to have their tools evaluated. Although this reality in no way minimizes the inherent value of independent security certification, it does indicate that whole categories of security tools useful in the development of Web applications will probably never undergo CC evaluation. Furthermore, there is an indication that the tools that are evaluated will not be sufficient in the variety of functions they perform to collectively provide all of the security functionality needed in sophisticated Web applications. Therefore, we are proposing a strategy for assessing tools that have not undergone and probably will not undergo CC evaluation.

If an available tool in a category has undergone, or is well on its way to achieving CC evaluation, that is the tool you should choose (by DoD policy mandate). However, if there are no CC-evaluated tools in a given category, we suggest you consider the following hierarchy of other possible security certifications, to determine whether a tool has attained any of these certifications.

This hierarchical lists progresses in descending order through a series of *next most desirable* certifications. As you research and assess tools, whenever two or more functionally comparable non-CC-evaluated tools are available, you can determine whether any has one of the other certifications on this list. If so, choose the candidate with the *most desirable* security certification.

This strategy is intended as a casual one for helping you narrow down options from multiple available technologies. It will be up to your DAA to decide whether a waiver of the CC evaluation requirement is warranted in the case of a particular product or technology. Still, the fact that a product has one or more of the other certifications may be useful in helping you to sell the waiver request to your DAA. Table C-1 provides details.

Table C-1. Hierarchical List of Possible Information Assurance Certifications

NTISSP 11 REQUIRED (MOST DESIRABLE)	
1a	COTS: CC validation under NIAP
1b	GOTS: Certification by NSA in accordance with NTISSP 11
NEXT MOST DESIRABLE (when no acceptable NIAP-validated product exists) <i>(in descending order of preference)</i>	
2	CC validation not yet appearing on the NIAP validated products list, by a certification authority in a country included in the CC recognition arrangement
3	Included on NIST's Federal Information Processing Standard (FIPS) 140 validated products list
4	Recent evaluation (within the past two product releases) under the TCSEC, by the National Computer Security Center (NCSC) in its Trusted Product Evaluation Program (TPEP), or by NIST
5	Included on NSA Advanced Encryption Standard (AES) Validation List or Triple

	Data Encryption Standard (DES) Validation List
6	Accredited according to DoD Information Technology Security Certification and Accreditation Process (DITSCAP) or DCID 6/3 C&A process
7	Included on an approved IA products list of a C/S/A
8	Accredited by a civilian intelligence, diplomatic, or law enforcement agency or included on an approved IA products list of such an agency, or both.
9	Evaluated recently (within the past two product releases) under the Information Technology Security Evaluation Criteria (ITSEC)
10	Accredited for North Atlantic Treaty Organization (NATO) use, or on a NATO-approved IA products list
11	Certified by a widely recognized commercial security organization, such as ICSA Labs
12	Granted an award by one or more respected information security or defense/military trade publications
13	Given outstanding reviews by one or more respected information security or defense/military trade publications

C.2 THE TOOLS

The tools listed in Table C-2 are identified for information purposes only; inclusion of a tool does not constitute an endorsement of that tool. No technical assessment was done for any of these tools, beyond reading the product literature (or, in the case of public domain tools, the high-level technical description).

Table C-2. List of Third-Party Security Tools

Main Function	Supplier	Tool Name	Description	Categ.	URL
Signature validation	KyberPass	Validation TrustPlatform	Digital signature validation and OCSP-based certificate validation	COTS	http://www.kyberpass.com/products/validation_trustplatform.html
Signature validation	Gilian Technologies	G-Server	Digital signature of Web content; validates signature before serving content to browser	COTS	http://www.gilian.com/gserver.html
Signature validation	Gilian Technologies	ExitControl	Verification of Web content validity before serving it to browsers	COTS	http://www.gilian.com/gserver.html
Secure Sockets Layer (SSL)	RSA Security	BSAFE SSL-C	Software developer kit (SDK) of C components for integrating SSL into C applications; includes support for X.509 certificates for server and client authentication and certificate verification interoperable with Netscape, Microsoft, and VeriSign certification schemes	COTS	http://www.rsasecurity.com/products/bsafe/slc.html
SSL	RSA Security	BSAFE SSL-J	SDK of Java components for integrating SSL into Java applications; functionally comparable to BSAFE SSL-	COTS	http://www.rsasecurity.com/products/bsafe/slj.html

			C		
SSL	Baltimore Technologies	KeyTools SSL	Crypto toolkit for implementing SSL documents; works in conjunction with KeyTools Pro	COTS	http://www.baltimore.com/keytools/ssl/index.asp
SSL	Certicom	SSL Plus	SDK for implementing SSL 2.0, SSL 3.0, and TLSv1.0 functionality in C desktop applications; supports two-way authentication (client-server and server-client), RSA and ECC SSL ciphers, X.509 certificate handling, and DSA certificate signing; fully interoperable with Netscape Navigator/Communicator, Microsoft Internet Explorer SSL implementations	COTS	http://www.certicom.com/products/ssl_plus/ssl_plus_desktop.html
SSL	Certicom	SSL Plus for Java	SDK for implementing SSL functionality in Java applications; comparable to SSL Plus (for C), plus supports Sun's Java Secure Sockets Extension (JSSE) API and is fully interoperable with JDK 1.2 and JDK 1.3	COTS	http://www.certicom.com/products/ssl_plus/ssl_plus_java.html
SSL	OpenSSL Project	OpenSSL	Toolkit based on Eric A. Young's and Tim J. Hudson's SLeay library, for implementing SSLv2, SSLv3, and TLSv1 protocols in applications; includes cryptographic library	Open Source	http://www.openssl.org
SSL	Stunnel.org	Stunnel	SSL tunneling of TCP and application-layer protocols not otherwise supported by SSL	Open Source	http://www.stunnel.org/ , see also: http://www-106.ibm.com/developerworks/security/library/s-stun.html?dwzone=security
SSL hardware accelerator	NCipher Corp. Ltd.	nFast SSL accelerator	Improves security and increases server throughput in applications using SSL, such as secure Web servers, authenticated access to intranets and extranets, digital signatures, and secure messaging. Several nCipher cryptographic accelerators are FIPS 140-1 validated: nFast nF75KM 1C, nF150KM 1C, nF300KM 1C, nFast nF75CA 00, nF150CA 00, nF300CA 00, nFast nF75KM 00, nF150KM 00, and nF300KM 00; also nForce 400 SCSI and nForce 150 SCSI; nForce 300 PCI and nForce 150 PCI; nForce 300 SCSI, nForce, 150 SCSI, and nForce 75 SCSI; and nForce 300, nForce 150, and nForce 75.	COTS	http://www.ncipher.com/nfast/
SSL hardware accelerator	Chrysalis-ITS Ltd.	Luna XL and XLR	Provides high-performance hardware-based key management and cryptographic acceleration for Web	COTS	http://www.Chrysalis-ITS.com/trusted_systems/luna_xl.htm

			servers that do SSL transaction processing (the XLR is a 1U rack-mount system). FIPS 140-1 validated		
Digital signature hardware engine	Chrysalis-ITS Ltd.	Luna XP plus	Provides hardware-accelerated signing, secure key management, and signature validation for high-volume transaction applications, such as transaction coordinators and OCSP (Online Certificate Status Protocol) responders. FIPS 140-1 validated	COTS	http://www.Chrysalis-ITS.com/trusted_systems/luna_XPplus.htm
Certificate validation	CertCom	CertValidator and CertValidator Toolkit	OCSP repository and responder for OCSP-based certificate validation	COTS	http://www.certco.com/certvalidator.shtml
Certificate validation	KyberPass	Validation TrustPlatform	OCSP-based certificate validation and digital signature validation	COTS	http://www.kyberpass.com/products/validation_trustplatform.html
Certificate validation	ValiCert	Validation Authority	X.509 certificate validation middleware	COTS	http://www.valicert.com/products/validation_authority.html
Certificate handling	Entrust CygnaCom	X.509 Certificate Path Development Library	For Microsoft Windows 32-bit Operating Systems: API for implementing X.509 certificate path development from a specified entity to a trusted root. After generation by the API, the certificate path is returned to the API user for verification; verification itself is performed by a program outside of the API and may consist of signature verification, certificate revocation list checks, policy verification, and any other verification desired.	COTS	http://www.entrust.com/entrustcygnacom/products/index.htm
Certificate handling	RSA Security	BSAFE Cert-C	Certificate handling SDK of C software libraries, sample code, and documentation to ease integration into C applications of PKI-based cryptographic procedures for handling X.509 certificates	COTS	http://www.rsasecurity.com/products/bsafe/certc.html
Certificate handling	RSA Security	BSAFE Cert-J	Certificate handling SDK of Java software libraries, sample code, and documentation to ease integration into Java applications of PKI-based cryptographic procedures for handling X.509 certificates	COTS	http://www.rsasecurity.com/products/bsafe/certj.html
Single sign-on (SSO), policy mgmt.	KyberPass	Web Access TrustPlatform	Adds centralized Web access policy management, authentication and authorization using OCSP and SSL to Web applications without a developer toolkit or custom development	COTS	http://www.kyberpass.com/products/web_access.html
SSO, policy mgmt.	RSA Security	ClearTrust	Centralized Web access policy management, SSO, delegation,	COTS	http://www.rsasecurity.com/products/cleartr

			authorization, and self-audit		ust/index.html
SSO, policy mgmt.	BioNetrix	Authenticatio n Suite 4.1	Authentication management infrastructure (software) for unified management of combination of advanced authentication technologies, including smart cards, tokens, and biometrics; implements policy-based client-server, legacy, and Web based applications	COTS	http://www.bionetrix.com/products.html
Kerberos	CyberSafe	TrustBroker Web Authenticatio n Solutions	TrustBroker Web Agent plus C, C++ and Java application development toolkit of functions for Web based authentication using Kerberos	COTS	http://www.cybersafe.ltd.uk/products_solutions_was.htm
Kerberos	CyberSafe	Kerberos Database Solutions	Adds Kerberos authentication to Oracle and Sybase applications	COTS	http://www.cybersafe.ltd.uk/products_solutions_dss.htm
Pluggable authentication module (PAM)	PADL Software Pty Ltd	pam_ldap	Provides means for users of UNIX (incl. Solaris) and Linux workstations to authenticate against LDAP directories and to change their passwords in the directory; uses the PAM API defined in OSF DCE RFC 86.0; supports SSL/TLS for session encryption and strong authentication; supports Netscape Directory Server password policies and directory-based access authorization	Open Source	http://www.padl.com/OSS/pam_ldap.html
PAM	Students at University of Michigan	NI_PAM	PAM-like pluggable authentication modules to NT's GINA authentication module	Open Source	http://www.citi.umich.edu/projects/singlesignon/poster2.html ; http://www.citi.umich.edu/u/itoi/demo19971016/poster2.html
PAM	A.G. Morgan	Pluggable Authenticatio n Module for Apache	PAM implementation for Apache Web server, comprises various authentication modules and programs from various developers	Open Source	http://pam.sourceforge.net/mod_auth_pam/ , see also: http://www.kernel.org/pub/linux/libs/pam/modules.html
One-time password (OTP)	RSA Security	SecureID product suite	ACE/Server enterprise authentication server, ACE/Agent for Web, SecureID tokens and smart cards, agents for other types of systems, etc.	COTS	http://www.rsasecurity.com/products/secureid/index.html
OTP	Naval Research Laboratory	One-Time Passwords In Everything (OPIE)	Implementation of the OTP standard specified in IETF RFC 1938; derived in part from the BSD UNIX and in part from the Bellcore's S/Key, plus enhancements developed by NRL	GOTS	http://www.inner.net/pub/opie/
OTP	Bellcore	S/Key	The granddaddy of RFC 1760-compliant OTP generators (for UNIX)	Open Source	ftp://thumper.bellcore.com/pub/nmh/
OTP	Sandia National Laboratory	S/Key MD4 calculator	Calculates MD4 digests for use by S/Key	Open Source	ftp://ftp.cs.sandia.gov/pub/firewall/skey/
OTP	Harry	jotp (Java	Java applet implementation of S/Key	Open	http://www.cs.umd.e

Draft

	Mantakos	OTP Calculator)	OTP system	Source	du/~harry/jotp/
OTP	Markus Kuhn, Computer Laboratory, University of Cambridge	OTPW: One Time Password Login Capability	OTP generator called newpass plus two verification routines — otpw_prepare() and otpw_verify() — that can be added to programs such as login or ftpd on UNIX or Linux systems	Open Source	http://www.cl.cam.ac.uk/~mgk25/otpw.html
Biometrics	Bioscrypt	BIO-SDK	SDK for implementing APIs to Bioscrypt biometric devices and servers, and implementing functions such as reader configuration capability, authentication, and biometric data transfer. NSA approved Bioscrypt's Enterprise Reader, v.2.0.1.C1; Enterprise for Windows NT version 2.1.3 was NIAP CC certified.	COTS	http://www.bioscrypt.com/products/bio_sdk.shtml
Role-Based Access Control (RBAC)	NIST	RBAC reference implementation	Adds RBAC into existing UNIX and Windows NT operating systems	GOTS	http://csrc.nist.gov/rbac/#software
Application firewall	Ubizen	DMZ/Shield	Validates incoming (from browsers) Web requests for conformance with security policy; prevents execution of known attacks or administrator-configured unallowed Web requests	COTS	http://www.ubizen.com/c_products_services/3_ubizen_dmzshield/c331.html
Application firewall	Sanctum, Inc.	AppShield	Web application firewall that prevents the Web server from performing administrator-defined not allowed actions requested by browsers; also audits and notifies administrator of attempted violations	COTS	http://www.sanctuminc.com/solutions/appshield/index.html
Application firewall	eEye Digital Security	SecureIIS	Application firewall for Microsoft IIS Web server inspects all incoming requests from browsers and prevents any potentially damaging requests — whether received in encrypted or unencrypted transmissions - from being responded to by the server. Uses multiple security filters to inspect Web traffic received by the server for buffer overflows, parser evasions, directory traversals, and other attacks, and prevents server from responding to such traffic	COTS	http://www.eeye.com/html/Products/SecureIIS/index.html
Secure execution	NAI Labs	Generic Software Wrappers	Prototype software wrapping technology to increase the security and reliability of large software systems composed of standardized software components. Generic Software Wrappers intercept component interactions and bind them with additional functions that implement	Open Source (DARPA - funded R&D)	http://www.nai.com/research/nailabs/secure-execution/wrappers-darpa.asp

			practical security (e.g., restricting, filtering) and reliability (e.g., redundancy, crash data recovery) policies		
Code integrity	Macrovision	SafeWrap	Wrapper to protect Windows-based application code from being tampered with or reverse-engineered COTS	Open Source	http://www.macrovision.com/solutions/software/safewrap.php3
Code integrity	David Wagner, Tal Garfinkel	Janus	Tool for sandboxing untrusted applications within a restricted execution environment on Linux systems	Open Source	http://www.cs.berkeley.edu/~daw/janus/
Code integrity	NCipher	Secure Execution Engine (SEE) and CodeSafe Developer Kit	NCipher's SEE is a tamper-resistant hardware security module for executing sensitive Java and C/C++ programs securely. The CodeSafe Developer Kit is an SDK of APIs to develop trusted Java or C/C++ programs as NCipher Trusted Agents that will be securely loaded and executed on an nCipher hardware security module (HSM).	COTS	http://www.ncipher.com/technologies/see.html and http://www.ncipher.com/safebuilder/codesafe.html
Hash	Bokler Software Corporation	HASHcipher Library	SHA-1 cryptographic library for Windows application developers	COTS	http://www.bokler.com/hashcipher.html
Web content signature/ encryption	Appligent	SecurSign	Digital signature and 128-bit encryption of Adobe PDF documents by Web authors	COTS	http://www.appligent.com/newpages/secursign.html
Web content signature/ encryption	Lexign	ProSigner	Digital signature of Microsoft Word and Excel documents and Adobe PDF documents by Web authors	COTS	http://www.lexign.com/products/lexign_prosigner.htm
Web content signature/ encryption	Appligent	APCrypt	Server-based command-line applying standard Acrobat encryption to PDF documents	COTS	http://www.appligent.com/newpages/apcrypt.html
Web content time/date stamp	Appligent	DateD 2.0	Plug-in for Adobe Acrobat that adds dynamic date and time stamps or static messages to PDF documents	COTS	http://www.appligent.com/newpages/dated.html
Web content security filter SDK	IDRSI	CHX-I Developer Kit 1.9	Forces use of client-side SSL and 128-bit encryption on the server-side SSL. Implements application firewall actions on multiple WildFlag presences and lists WildFlag in the algorithm. Performs logical operations on multiple wilds (AND/OR). Provides packet payload control with insert and replace actions. Supports multiple pseudo wild possibilities: Character, String, and Remote. Implements optimized redirection service.	COTS	http://www.idrci.net/idrci_products.htm
Web content integrity protection (copy and defacement)	OmniSecure	HTTPprotect	Based on VP Disk Pro as the underlying core technology, HTTPprotect comes with customized configuration, streamlined installation scripts, and Web-based GUI designed specifically	COTS	http://www.omnisecur.com/products/http-prod-brief.pdf http://www.omnisecur.com/httpprotect.html

			to secure Web contents, CGI scripts, and data collected from on-line users via the Web.		
Web content integrity protection (copy and defacement)	Odyssey Technologies	WIGIL	Monitors Websites for changes. If a modification is detected, it alerts the administrator and automatically republishes the original (non tampered version) of the page.	COTS	http://www.odysseyte.com/Products/Wigil.html
Web content integrity protection (copy and defacement)	Odyssey Technologies	AssurePage	Comprises AssurePage Server and AssurePage Viewer, which enable Web content developers to add end-user authentication and integrity checking using Public Key Infrastructure (PKI) digital signature of Web content to prove to users that the content they view is in fact valid and has not been tampered with	COTS	http://www.odysseyte.com/Products/AssurePage.html
Web content integrity protection (copy and defacement)	Andreas Wulf Software	HTML Guard	Encrypts HTML source code, and disables the right mouse button of Windows PCs and the text selection and print functions of browsers that connect to the Web server, to inhibit copying (and defacement) of Web content	Open Source	http://www.aw-soft.com/htmlguard.html
Web content integrity protection (copy and defacement)	Andreas Wulf Software	WebExe	Converts a single HTML page or group of pages into a single, self-running EXE file with an integrated browser, providing the developer full control of the browser's functionality, including the appearance of menu bars, printing, and copying	Open Source	http://www.aw-soft.com/webexe.html
Web content integrity protection (copy and defacement)	Authentica	NetRecall	Toolset for controlling how Web content can be used after it is downloaded/accessed by users; includes Authentica policy server for managing and storing protection policies, distributing keys, managing client connections, and logging all Web content accesses; Web viewer client plug-in enables viewing of protected content; Content Manager supports Web author encryption of content, establishment of access policies, and tracking of access of protected content; Dynamic Protection Module for protecting Web content dynamically generated by the Web server	COTS	http://www.authentica.com/products/netrecall/how_it_works.asp
Web content integrity protection (copy and defacement)	Authentica	PageRecall	Implements page-level control over documents during and after delivery to recipients, controlling who can read the document, which pages they can read, when they can read the	COTS	http://www.authentica.com/products/pagerecall.asp

			document, and when read-access should expire. Centrally manages viewing rights and expiration, even after document distribution. Enables document creator to prohibit printing of all or part of the document, enables adding of a watermark to printable pages, and enables the "copy/paste" and "save as" functions to be disabled for the document.		
Web content integrity protection (copy and defacement)	ArtistScope	CopySafe	For Web sites on Windows NT, provides Web image encryption and a browser plugin to protect and restrict usage, and prevent copying	COTS	http://www.artistscope.net/copysafe/index.html
Encryption of data at rest	F-Secure Security Solutions	F-Secure FileCrypto for Desktops and Laptops — Kernel Mode	Encryption of files, not the entire hard drive; centralized key management and recovery by a administrator using F-Secure Policy Manager. Cryptographic Driver Version 1: NSA approved for AES and 3-DES; Cryptographic Service Provider DLL Version 1.1: NSA approved for 3-DES.	COTS	http://www.f-secure.com/products/filecrypto/desktop.shtml
Encryption of data at rest	F-Secure Security Solutions	F-Secure Policy Manager	Central management of file and network encryption, key management, antivirus protection, and other enterprise security functions	COTS	http://www.f-secure.com/products/policy-man/
Encryption of data at rest	Protegrity	Secure.Data	Data item-level encryption for database entries, with data access policy management	COTS	http://www.protegrity.com/The_Secure.Data_Suite.html
Encryption of data at rest	Application Security Inc.	DbEncrypt for Oracle and Microsoft SQL Server	Encrypts rows and columns in the database	COTS	http://www.appsecinc.com/corporate/protect.html
Encryption of data at rest	ERUCES Data Security	ERUCES Tricryption Engine Database Edition	High-volume database encryption and automated key management system, enabling centralized control over encryption keys used enterprise wide	COTS	http://www.eruces.com/default.asp?=-Products
Encryption of data at rest	ERUCES Data Security	ERUCES Tricryption Engine Key Host Edition	High-volume file encryption and automated key management system, enabling centralized control over encryption keys used enterprise wide	COTS	http://www.eruces.com/default.asp?=-Products
Encrypted data transfer	XYPRO	XYGATE File Encryption	Encryption of unstructured data files before transfer. Operates on Window, UNIX, Linux, IBM OS390, and Himalaya (Tandem/NSK) systems; key management through XYGATE KM; XYCRYPT 3.0 (used in all XYPRO encryption products) NSA approved and NIST FIPS 140 certified for 3-DES; NIST FIPS 140 certified for DSA and SHA1.	COTS	http://www.xypro.com/products/xygatefe.html
Encrypted data	Securit-e-doc	Securit-e-doc	Provides Web server-based one-time	COTS	http://securit-e-

transfer			use symmetric encryption (using Securit-e-doc's Secure Information Transport Technology [SITT]) to implement encrypted filesystem (called S-Doc) with full key management on Web server. Enables user access to encrypted files via interactive HTML interface from browser to server. SITT Cryptosystem Version 3.0: NSA approved for AES and 3-DES.		doc.com/product/pro.htm
Public Key Enabling (PKE) SDK	Certicom	Security Builder	Cryptography toolkit including APIs and C and Java components for integrating encryption, digital signatures, and related security mechanisms into applications. Government Solutions Edition Version10: NSA approved for AES and 3-DES.	COTS	http://www.certicom.com/products/securitybuilder/securitybuilder_feat.html
PKE SDK	RSA Security	BSAFE Crypto-C Version 5.2.1	PKE toolkit of cryptographic components and APIs in C, including DSA, AES, 3-DES, SHA-1, PKCS#11, and numerous other cryptographic and key management functions optimized for Intel Pentium and Sun SPARC platforms. NSA approved for 3-DES, FIPS 140-1 and 140-2 certified.	COTS	http://www.rsasecurity.com/products/bsafe/cryptoc.html
PKE SDK	RSA Security	BSAFE Crypto-J	PKE toolkit and APIs in Java, implementing the same functions as those provided in Crypto-C	COTS	http://www.rsasecurity.com/products/bsafe/cryptoj.html
PKE SDK	Baltimore Technologies	KeyTools Pro	PKE toolkit, in C++ and Java versions, including APIs and components for cryptographic and certificate handling, CRL distribution and checking, OCSP, LDAP, central policy control, smart card support, and cryptographic algorithms (including DSA, 3-DES, and SHA-1). Uses any JCE/JCA (Java Cryptography Extension/Java Cryptography Architecture) compliant cryptographic provider (including Sun's native JCE provider) or Baltimore's own JCE provider (KeyTools Pro)	COTS	http://www.baltimore.com/keytools/pro/index.asp
PKE SDK	Certicom	Trustpoint/C	SDK for adding PKI functionality to C and C++ server applications on Windows, Solaris, and Linux; includes support for X.509v3 certificate and CRL handling, PKCS#10, cryptographic algorithms (including DSA, 3DES), S/MIMEv2, and interoperability with several vendors' PKIs and CAs	COTS	http://www.certicom.com/products/trustpoint_toolkits/trustpoint_tool_c.html

PKE SDK	Certicom	Trustpoint/Java	SDK for adding PKI functionality to Java server applications on Windows, Solaris, and Linux; functions comparable to those in Trustpoint/C	COTS	http://www.certicom.com/products/trustpoint_toolkits/trustpoint_to_java.html
PKE SDK	MegaSign	Application Integration Toolkit	SDK for PKI-enabling new or legacy applications; includes C and C++ APIs, and library functions for X.509v3 certificate registration and lifecycle management (including CRL handling), cryptographic interfaces, LDAP and X.500 directory interfaces, HTTP support	COTS	https://www.megasign.nl/onsite/datasheets/toolkit/
General crypto SDK (symmetric and PK)	Bokler Software Corporation	TDEScipher Library	Symmetric cryptography toolkit for Windows application developers		
NSA approved for 3-DES; NIST FIPS PUB 46-3 compliant	COTS	http://www.bokler.com/tdescipher.html			
General crypto SDK (symmetric and PK)	Information Security Corporation	ISC Cryptographic Development Kits	Libraries of linkable cryptographic modules for adding encryption, digital signatures, and message authentication to applications, NSA approved for 3-DES	COTS	http://www.infoseccorp.com/products/cdks.htm
General crypto SDK (symmetric and PK)	XYPRO	XYGATE Encryption Software Developer Kit	SDK comprising the XYCRYPT library of cryptographic mechanisms for encryption-enabling applications; XYCRYPT 3.0 (used in all XYPRO encryption products). NSA approved and NIST FIPS 140 certified for 3-DES; NIST FIPS 140 certified for DSA and SHA1.	COTS	http://www.xypro.com/products/xygateencryption.html
Pseudorandom number generator	Secure Software	Entropy Gathering and Distribution System (EGADS)	System service and library for providing secure random numbers on Windows and UNIX systems; provides the same functionality on these platforms that /dev/random and /dev/urandom provide on Linux partly funded by DARPA CHATS program and evolved from DARPA-funded Yarrow pseudo-random number generator)	Open Source	http://www.securesw.com/egads.php
Pseudorandom number generator	Brian Warner	Entropy Gathering Daemon (EGD)	Perl substitute for /dev/random on systems that do not have a convenient source of random bits	Open Source	http://egd.sourceforge.net/
Web security infrastructure	Wipro Technologies	WebSecure	Enterprise Web security infrastructure: implements authentication (SSO), authorization, session management, encryption, non repudiation, audit, and automated security incident response	COTS	http://www.wipro.com/wiprowebsecure/products/overview.htm
Software development environment	TogetherSoft	Control Center	Application development environment and toolkit	COTS	http://www.togethersoft.com/products/controlcenter/index.jsp

Software development environment	Rational	eXtended Development Environment (XDE)	Application development environment and toolkit	COTS	http://www.rational.com/products/xde/index.jsp
Java security	The Jakarta Project	Jakarta Turbine	Servlet-based framework for developing secure Java applications based on a single-entry point program model	Open Source	http://jakarta.apache.org/turbine/ See also http://www.javausergroup.at/events/turbine.pdf
Common gateway interface (CGI) security	Stanford University Linear Accelerator Center	cgi-wrap	CGI security wrapper that performs simple checking on input to CGI scripts, imposes resource-usage limits on processes created by CGI scripts, and kills stalled processes	Open Source	http://www.slac.stanford.edu/slac/www/tool/cgi-wrap/doc/
CGI security	Nathan Neulinger	CGIWrap	Gateway program that enables more secure user access to CGI programs on an HTTPd server than provided by the server itself by making certain that a CGI script runs with the permissions of the user who installed it, not with the permissions of the server. (Use this wrapper with caution: if an attacker manages to execute commands under a valid user's username, he can delete or modify all of that user's data and indeed the user's account itself.)	Open Source	http://cgiwrap.unixtools.org/
CGI security	Steven Grimm	Uncgi	Front end for processing queries and forms from the Web; decodes all form fields and inserts them into environment variables for easy perusal by a C program (Perl script, etc.) then executes whatever other program is specified by that program/script. Eliminates the need to write or find and use routines to translate URL-encoded values in HTML form fields into a format understandable by the program/script processing those data; no need to write or use different routines for handling forms received via GET versus POST.	Open Source	http://www.midwinter.com/~koreth/uncgi.html
Perl security	mod_perl	Apache::Taint Request	Applies Perl "tainting" rules to HTML output	Open Source	http://www.modperlcookbook.org/code.html
Python security	Unknown	PyChecker	Source code error scanning for Python programs	Open Source	http://pychecker.sourceforge.net
C & C++ security	Todd Austin, Wisconsin Multiscalar Group, University of Wisconsin-Madison	Safe C Compiler	Optimizing C-to-C compiler, which implements the extended pointer and array access semantics needed to provide efficient, reliable, and immediate detection of memory access errors	Open Source	http://www.cs.wisc.edu/~austin/scc.html

Draft

C & C++ security	Greg McGary	Bounds checking extensions	Add fine-grained bounds checking to GCC's C and C++ compiler front ends	Open Source	http://gcc.gnu.org/projects/bp/main.html
Buffer overflow prevention	AUSCERT	overflow_wrapper.c	Wrapper for limiting exploitation of programs that have command-line argument buffer overflow vulnerabilities	Open Source	ftp://ftp.auscert.org.au/pub/auscert/tools/overflow_wrapper.c
Buffer overflow prevention	Paul Szabo	sec_wrapper.c	Wrapper (UNIX setuid program) for preventing command-line or environment variable buffer overflow by checking the lengths of arguments or environment variables (does not check values of those arguments or environment variables)	Open Source	http://www.maths.usyd.edu.au:8000/u/psz/du/sec_wrapper.c
Buffer overflow prevention	Immunix.org	StackGuard	Memory integrity checking enhancement to GCC C/C++ compiler that augments generalized bounds checking to detect and prevent buffer overflows on stacks (but not on heaps) while requiring no changes to program source code. When compiled into a program, StackGuard detects in real time attempts to exploit stack smashing vulnerabilities, raises an intrusion alert, and halts the program before the exploit can be accomplished.	Open Source	http://immunix.org/stackguard.html
Buffer overflow prevention	RST Software Security Group	ITS4	Helps automate source code review for security by statically scanning C and C++ source code for potential security vulnerabilities. It is a command-line tool that works across UNIX systems and on Windows platforms running CygWin.	Open Source	http://www.rstcorp.com/its4
Buffer overflow prevention	Bell Labs	Libsafe	Middleware software layer that intercepts all function calls made to library functions that are known to be vulnerable to buffer overflow attacks. Operates on executing programs, so does not require access to source code of defective programs, recompilation, or off-line processing of binaries. For each vulnerable call, Libsafe substitutes a safe version of the corresponding function that implements the original functionality, but in a manner that ensures that any buffer overflows are constrained within the current stack frame, thus preventing attackers from smashing (overwriting) the return address and hijacking control flow of the running program.	Open Source	http://www.bell-labs.com/org/11356/libsafe.html

Extensible Markup Language (XML) security	IBM alphaworks	XML Security Suite	Provides digital signature, encryption, and access control for XML documents	COTS	http://www.alphaworks.ibm.com/tech/xmlsecuritysuite
XML security	Aleksey Sanin	XMLSec	XML security library in C for implementing standards-based XML SignatureXML Encryption, Canonical, and Exclusive Canonical XML	Open Source	http://www.aleksey.com/xmlsec/
XML security	Baltimore Technologies	KeyTools XML	Crypto toolkit for securing XML documents; works in conjunction with KeyTools Pro	COTS	http://www.baltimore.com/keytools/xml/index.asp
XML security	Westbridge	XML Application Firewall	Enforces RBAC access control rules for XML objects		http://www.westbridge.tech.com/appfirewall.html
Security code sample (illustrative)	Novell Corporation	mutual.c	Demonstrates mutual authentication with an LDAP server via SSL	COTS	http://developer.novell.com/ndk/doc/samplecode/cldap_sample/mutual.c.html
Common objection request broker architecture (CORBA) security	Adiron	ORBAsec SL3 CORBA	Middleware implementation of standard CORBA Common Secure Interoperability Version 2 (CSIV2) protocol, with APIs to credentials-handling and SSL/TLS functionality	COTS	http://www.adiron.com/ORBAsec3.html
CORBA security	Object Security	MICOsec	Middleware implementation of standard CORBA Security Services (CORBAsec) Level 2 Version 1.7, based on the MICO Object Request Broker (ORB) and standard SSL	COTS	http://www.objectsecurity.com/micosec.html
Browser input validation [see note]	WhiteHat Security	WhiteHat Arsenal	Browser security toolset that adds a series of input validations with automatic correction of HTML forms to eliminate dangerous input; also adds logging of security-relevant events in the browser	COTS	http://www.whitehats.ec.com/html/wharsenal.html
Browser digital signature	RSA Security	Keon e-Sign	Browser plug-in for digital signature of Web forms	COTS	http://www.rsasecurity.com/products/keon/datasheets/dskeonesign.html
Browser digital signature	Netscape	Form Signing	Digital signature of Web forms for Netscape browsers	COTS	http://developer.netscape.com/tech/security/formsign/formsign.html
Browser digital signature	Lexign	Web Signer	Digital signature of Web forms	COTS	http://www.lexign.com/products/lexign_web signer.htm
PKE of legacy applications	Celocom	eAccess Server	Tool for PKE of existing applications (including legacy applications) with minimal reprogramming of those applications. The eAccess Server is an SSL/TLS gateway with extended PKI capabilities that extends PKI services (strong encryption, revocation control and PKI-based access control) to	COTS	http://www.celocom.com/web/celo/d.asp?p=682

			Web portals and existing legacy client-server applications. The eAccess Server provides a platform for establishing a generic SSL/TLS server-side tunnel that can then be applied to any static TCP-protocol, such as HTTP, Telnet, POP3, and ICA, used between clients and servers.		
--	--	--	--	--	--

NOTE: Use browser input validation only in conjunction with server input validation, never as a substitute for server security tools. Browser input validation can be useful as a first cut, to reduce the number of possible violations passed to the server. The server must validate all browser input regardless, in case browser input validation was bypassed or the data output by the browser was tampered with en route to the server.

APPENDIX D: PROGRAMMING LANGUAGE SECURITY

D.1 C AND C++

D.1.1 GENERAL RULES OF THUMB FOR SAFE C PROGRAMMING

1. Check all functions for valid returns.
2. Verify the validity of all environment variables before using them.
3. Set the *PATH* environment variable to a known value.
4. Use full (not relative) pathnames in all commands.
5. On UNIX systems, do not use `setuid` on a whole program. Use `setuid root` only for the smallest part of the program that needs to have *root* privilege, and then as soon as that part of the program has completed its *root*-privileged task, `setuid` back to user.
6. Strip all binaries.
7. Log all usernames, file accesses, and related items.
8. Avoid common semantic errors and typos, such as using “=” when you mean “==”.
9. C does not implicitly provide exception handling. Therefore, write your programs to explicitly handle critical errors. Do not rely on the programming language to do so.

D.1.1.1 Type Declaration and Checking

Declare types strictly. Whenever possible, use `enum` to define enumerated values; do not just define a character type or integer type (*char* or *int*) with special values. This is particularly true for values in switch statements, whereby the compiler can be used to determine whether all legal values have been covered.

When a value cannot be negative, use unsigned types if possible. Be aware that when a signed *char* with its high bit set is saved in an integer, the result will be a negative number; this may result in an exploitable vulnerability. Unless absolutely impossible, when dealing with character data that may have values greater than 127 (0x7f), use *unsigned char* instead of *char* or *signed char* for buffers, pointers, and casts.

Validate all input. Check all input arguments for validity — not just the C type, but also program-specific type information. For example, if an input argument is supposed to be an executable file, check that the input file is an executable and that the user has permission to run the file.

D.1.1.2 Memory Allocation

C is unable to detect and prevent improper memory allocation. To avoid buffer overflows, developers must do their own memory management (e.g., using `malloc()`, `alloc()`, `free()`, `new`, and `delete`). Errors in memory management may result in security vulnerabilities. Most significant, programs that erroneously free memory that should not be freed (e.g., because it has already been freed) may immediately crash or be exploited by an attacker to cause arbitrary code to be executed. Using the incorrect call in C++ (e.g., mixing `new` and `malloc()`) may have similar results.

There are various tools available to address this problem (such as *Electric Fence* and *Valgrind*; see Appendix C). If unused memory is not freed for example, the unused memory may accumulate until the program stops working.

Most C programs also tend to set arbitrary limits on array sizes. Code that does not remember array bounds will typically crash. However, if a smart hacker corrupts the system's stack, he may use it to execute functions such as `system` instead of crashing (see Section D.1.2.1).

D.1.2 BUFFER OVERFLOW IN C AND C++

One of the most common security vulnerabilities, buffer overflows run rampant in many of today's applications. Surprisingly, this problem is not new, and in many cases it has arisen in the same operating systems and applications for decades (such as some UNIX variants). The frequency with which this type of problem is being discovered and exploited in mission critical software has grown significantly within the past decade. This is not because the problems did not exist, but because this type of attack required a higher level of sophistication than the average attacker possessed. Today, with cookie-cutter instructions on how to take advantage of these problems, and the shear increase in the number of attackers, minimal expertise is required.

A buffer overflow occurs when a piece of data is copied into a location in memory, one not large enough to hold the piece of data. The copying succeeds, however, and memory outside of the boundary of the target memory is written over. Variables in a program are allocated either on the programs stack or on the programs heap. Therefore, it is common to hear the terms *stack overflow* and *heap overflow*. Both types of overflows are possible to exploit, but the stack overflow is in many cases much easier.

Most buffer overflow problems in C and C++ originate in the standard C library, specifically in string operations that do no argument checking. Even if the application itself is written in another language that is not vulnerable to buffer overflows, because so many runtime library routines and Application Program Interfaces (APIs) used by most applications are likely to be written in C or C++, be careful to select only those routines/APIs that are not likely to cause buffer overflows.

Also, explicitly write all routines in the application to automatically compare the size of each input data string with the size of the buffer allocated to receive that string. If received data exceed the size of the allocated buffer, write the routine to

- Truncate or reject any data that exceeds the size of the buffer or
- Halt the operation and/or
- Return a warning message to the user.

D.1.2.1 Stack Smashing Vulnerabilities

Stack smashing is the most dangerous buffer overflow attack, particularly if the stack being attacked is running in privileged mode. The best way to avoid stack smashing is to implement nonexecutable stacks. This will prevent an attacker from being able to write and execute malicious code on the program stack. Several operating systems have nonexecutable stack patches, including Solaris. Other operating systems are inherently designed to allow only nonexecutable stacks.

Nonexecutable stacks can be defeated in programs that contain both a stack overflow and a heap overflow, because the stack overflow can be exploited to cause the program to jump to malicious code placed in the heap.

NOTE: Nonexecutable stacks may introduce some performance degradation. In pointer-intensive programs and other realtime programs where speed is critical, use of nonexecutable stacks may hamper adequate performance. When writing such programs, if you must use executable stacks, be particularly careful to avoid using buffer-overflow inducing calls, functions, and constructs.

D.1.2.2 Lack of Automatic Bounds Checking

C and C++ were designed intentionally not to do bounds checking automatically. This is because bounds checking adds processing overhead, and C and C++ were designed to favor efficiency over other consideration of language design. As a result, C programmers must compensate by writing their programs to explicitly perform bounds checking and by avoiding the use of unsafe C/C++ calls and functions.

With the increase in CPU performance in modern computers, argument and bounds checking no longer represents a drain on program efficiency. Indeed, any processing overhead these checks add to program execution will probably be undetectable.

D.1.2.3 Safe Alternatives to Dangerous Calls

Instead of using the following dangerous calls, functions, and library routines, use their safe alternatives, (shown in Table D-1). Most of these calls are dangerous because they can cause buffer overflows. In some cases, they may introduce other vulnerabilities.

NOTE: If there is no safe alternative to a dangerous call, either avoid the call completely and implement the desired functionality in another way; or be sure to

explicitly and correctly allocate adequate-sized buffers, and to include code in your program to perform bounds checking when using such calls.

Table D.1. Safe Alternatives to Dangerous Calls

Dangerous Calls	Safe Alternatives
gets()	fgets(buf, size, stdin), making sure allocated buffer is at least as big as specified
scanf sscanf() fscanf() vscanf() vsscanf() vfscanf()	fgetc, checking buffer boundaries when using fgetc in a loop or Use precision specifiers, and avoid “%” in formatting or Do your own parsing, and avoid “%” in formatting <i>NOTE: Do not use any call in the scanf() family to send data to a string without controlling maximum length.</i>
chmod()	fchmod()
chown()	fchown()
chgrp()	fchmod() or fchown()
strcpy()	strncpy(), making sure allocated buffer is at least as big as specified. When using strncpy(), you must explicitly NULL to terminate the string, because NULL termination will not occur if the length of the source buffer is larger than or equal to the size specified.
Wscopy _mbscopy	
strcat()	strncat(), only if absolutely necessary, and with caution
Wscat _mbsncat wcsncat _mbsncat	Do not use.
sprintf()	snprintf(), if supported and a safe implementation, and making sure allocated buffer is at least as big as specified or <i>glib</i> library alternative: g_snprintf() or Java alternatives: jio_fprintf, jio_snprintf, jio_vsnprintf or use precision specifiers
Vsprintf swprintf	Do not use.
system()	exec() Or place back slashes before any characters that have special meaning to the system shell before calling system()

shell() exec*p exec**	Do not use.
CopyMemory	Do not use.
getopt()family	Susceptible to overflows of internal static buffers. Always set the threshold for the length of inputs to be passed to <code>getopt()</code> and always do the necessary bounds checking.

*NOTE: Even some safe versions of system calls (e.g., `strncpy()` instead of `strcpy()`) are not completely safe. They sometimes leave strings unterminated or encourage subtle off-by-one bugs. Refer to David A. Wheeler’s *Secure UNIX Programming HOWTO*, “5.2 Library Solutions in C/C+,” for specific examples of such problems.*

In addition, we offer some caveats when using the calls found in Table D-2.

Table D-2. Safe Alternatives and Caveats

Dangerous Calls	Safe Alternatives
streadd() strecpy()	Allocate output buffer four (4) times larger than input buffer.
getenv()	Never assume the environment variable has a particular length.
realpath()	Allocate results buffer of <code>MAXPATHLEN</code> and make sure the length of the pathname to be input does not exceed <code>MAXPATHLEN</code> .
strtrns()	Manually check that the destination buffer is at least as big as the source string
Syslog	Do not use to log application events (it does not check file system environment variables for validity). Also truncate all string inputs to reasonable length before passing them to <code>syslog</code> .
Getopt getopt_long getpass	Susceptible to overflows of internal static buffers. Truncate all string inputs to reasonable length before passing them to <code>getopt</code> , <code>getopt_long</code> or <code>getpass</code> — and do necessary bounds checking
Getc fgetc() getpass() getchar() read	Susceptible to overflows of internal static buffers Truncate all string inputs to reasonable length before passing them to <code>getopt</code> , <code>getopt_long</code> or <code>getpass</code> Also, check buffer boundaries if using <code>getc</code> or <code>read</code> in a loop — and do necessary bounds checking.
Bcopy memcpy strncpy strcadd	Make sure that the allocated buffer is at least as big as specified.

streadd() strecpy() vsnprintf	
strlen()	Use only if you can ensure that there will be a terminating <i>NIL</i> character.
getwd(3)	Send a buffer at least <i>PATH_MAX</i> bytes in length
popen()	Place back slashes before any characters that have special meaning to the system shell before calling <code>popen()</code> .

NOTE: C/C++ `gets()` and stack overflow issues are present not only in UNIX, but also in Windows and MacOS. There have also been instances in which overflows have occurred due to unsafe usage of the `memcpy()` function. This may occur when the length specified to `memcpy()` function can be manipulated by an outside source. For this reason, always ensure that the length is not larger than the memory structure being copied into.

D.1.2.4 Tools for Detecting and Preventing Buffer Overflows

Two kinds of scanning tools have proven effective in helping to find and remove buffer overflows from C and C++ code. They are *static tools*, in which the code is considered but never run, and *dynamic tools*, in which the code is executed to determine its behavior.

Many static tools do little more than automate the `grep` commands to locate instances of problematic functions in source code. Thus help extract from a large program of tens or hundreds of thousands of lines of code the few hundred potential problems.

More effective static tools use data flow information to determine which variables affect which other variables, helping zero in on the truly problematic functions (versus the false positives) initially identified with a `grep`-based tool. The problem with the data flow approach is that it sometimes fails to flag calls that could cause problems.

Dynamic tools look for potential problems in code as it runs. One dynamic analysis approach is fault injection, in which the program is instrumented in a way that allows the tester to experiment with it, running what if scenarios against the code and watching the results. FIST is a fault injection tool that has been used to locate potential buffer overflows (see Appendix C).

Some combination of dynamic and static analyses is probably best for flagging buffer overflow vulnerabilities in code. Research is being done to determine which combination is most effective.

Use a compiler (or compiler add-on tool) that performs array bounds checking for C programs, such as the tool available for GNU C Compiler (GCC). Use of such a compile-time tool will prevent all buffer overflows, including all overflows in heaps and stacks.

Stackguard, ITS4, and Libsafe (see Appendix C) are just three open source developer tools that have proven very useful for detecting and preventing buffer overflows. Rational's Purify (see Appendix C) is a commercial memory integrity checking tool that is designed to protect against both stack and heap overflows. However, unlike its open source counterparts, Purify has been observed to make a significant performance impact on the programs it protects, which may make its use in production code impractical.

Whatever tools are used, the best approach to eliminating buffer overflow vulnerabilities is to protect the entire operating system with those tools. That action will ensure that not only the application program but also all the libraries called by it are protected.

D.1.2.5 Additional Solutions to Buffer Overflow

For additional specific solutions to C and C++ buffer overflow, please refer to David A. Wheeler, *Secure Programming for Linux and UNIX HOWTO*, "5.2 Library Solutions in C/C++" and "5.3 Compilation Solutions in C/C++" (see Appendix B). Also refer to Appendix C for specific buffer overflow detection and prevention tools.

D.2 VISUAL BASIC

Microsoft needs to publish security patches to Visual Basic fairly frequently. We strongly recommend that you use Java instead of Visual Basic.

D.3 JAVA

NOTE: Extensive information about Java's inherent security features is available in books and on the Web. Several references to such information appear in Appendix B.

Java code intended for use on the client runs in a different environment, under a different trust model, than does code on the server. There are common requirements, however, whether the Java code runs on the client or server. Input from untrusted sources should always be checked and filtered. Java code that inherits methods from parents, interfaces, or parents' interfaces also inherits vulnerabilities in those inherited methods. For this reason, it is critical that the developer use inheritance with caution.

The following are some specific guidelines for secure Java development:

- *Use a class loader that will enforce accessibility modifiers at run time.* The Java Virtual Machine (JVM) cannot be relied on to enforce the accessibility modifiers (such as private ones) in applications (versus applets) at run time. Run time enforcement of accessibility modifiers depends on which class loader is used to load the class requesting the access; a class loaded with a trusted class loader (including the null and primordial class loader), will return *TRUE* after an access check. By contrast, applets (e.g., appletviewer or browser) will not be loaded by a trusted or null class loader.

- *Do not depend on initialization to prevent allocation of uninitialized objects.*
- *Do not rely on package scope for security.* Although a few classes (e.g., java.lang) are closed by default, and some JVMs allow developers to close other packages, the majority of Java classes are not closed. They enable attackers to introduce new classes inside existing packages and use the new classes to access resources that were previously protected.
- *Minimize privileges and signed code.* Minimize or avoid altogether the use of special permissions and signed code. Strive to write programs that need only the permissions provided by the Java sandbox. Specific guidelines on use of privileged code follow.
- *Place all signed code in a single archive file.* To protect signed code from a mix-and-match attack (in which the attacker constructs a new applet or library that links existing signed classes with new malicious classes, or links signed classes that are not intended to be used together), use a single archive to store signed code. Note that existing code-signing systems provide inadequate protection against mix-and-match attacks. Use of a single archive will reduce their likelihood.
- *Globally protect packages:* Whenever possible, globally protect a package against package-insertion and package-access by an untrusted code.
- *Support object for sensitive information:* To ensure that sensitive information is explicitly cleared as soon as possible, store the sensitive information such as credentials in mutable data types, such as arrays, rather than in immutable objects such as strings. Do not rely on the Java platform's automatic garbage collection, which does not guarantee timely reclamation of memory. Clearing sensitive information as soon as possible makes it much harder for a hacker to target a heap-inspection attack from outside the virtual machine. Follow this caution: Do not embed secrets (cryptographic keys, passwords, or algorithm) directly in code or data. Hostile JVMs can be used to view the secrets stored in code and data. Obfuscation should not be used as a security approach.
- *Never return a reference to an internal array that contains sensitive data:* Even if an array contains immutable objects (e.g., strings), a copy — and not the original — should be returned, to prevent the calling code from changing the strings contained in the array. Instead of passing back the array, make a copy of the array and return that copy.
- *Reduce the scope of methods and fields:* Determine whether package-private members could be made private, whether protected members could be made package-private/private, and so on.
- *Make methods private.* Unless there is a good reason to make methods public, make them private. If a method must be public, the reason should be documented, and the method should include mechanisms to protect itself from the potential effects of tainted data it may receive.

- *Scrutinize native methods:* Examine all native methods for what they return, what they accept as parameters, whether they bypass security checks, whether they are public or private, and whether they contain method calls that bypass package boundaries (and thus package protection).
- *Do not use inner classes:* When translated into byte code, inner classes become accessible to any class in the package. Furthermore, the enclosing class's private fields become non private to permit access by the inner class.
- *Do not use public fields or variables:* Declare all fields and variables as private, and provide accessors (which can perform security checks) to them to limit their accessibility. In addition, include a security check in any public method that has access to and/or modifies any sensitive internal states, or both.
- *Do not use static field variables:* Static field variables are attached to the class itself, not to a class instance. A class can be located by any other class, thus making it possible for other classes to locate static field variables. That capability makes the static field variable difficult to secure.
- *Make objects immutable:* If possible, make objects immutable. Otherwise, make them clonable and return copies. Objects such as arrays, vectors, and hash tables are not immutable. That means the calling code can change the contents of these objects, with adverse security implications. Immutable objects cannot be changed by calling code. They also improve concurrency because no locking is needed. (See "Support object reuse for sensitive information", for an important exception to this rule.)
- *Never return a mutable object to potentially malicious code:* Arrays are mutable even when their contents are immutable. Do not return references to internal arrays that include sensitive data.
- *Never store user-supplied mutable objects or arrays directly:* Do not use a user-written (versus developer-written and tested) cloning routine. Constructors and methods that accept mutable objects, such as arrays of sensitive objects (e.g., public keys), should clone those mutable objects and arrays before saving them internally. These constructors and methods should not directly assign the array references to an internal variable of a similar type. If they do, a user who uses the constructor or method to create an object (including an immutable object) may be able to accidentally change the internal state of that object when making changes to an external array.
- *Make all classes and methods final:* Unless there is a good reason not to, make every class or method final to prevent hackers from extending it. It is true that finalizing classes and methods makes them impossible for the developer to later extend. But this is a trade-off that is worth the added security it provides. Do not use non final public static variables.

- *Make classes unclonable:* To prevent an attacker from using Java's object-cloning mechanism to instantiate a class without running any of its constructors, define the following method in each class:

```
private final Object clone()
    throws
        java.lang.CloneNotSupportedException
{
    throw new
        java.lang.CloneNotSupportedException()
    ;
}
```

If a class absolutely must be clonable, define a clone method and make it final, or add the following method:

```
private final void clone()
    throws
        java.lang.CloneNotSupportedException
{
    super.clone();
}
```

- *Make classes unserializable:* To prevent attackers from viewing the internal state (including private portions) of objects, add this method to the classes in the program:

```
private final void
writeObject(ObjectOutputStream out)
    throws java.io.IOException {
    throw new java.io.IOException("Object
cannot be serialized");
}
```

Even if serialization is considered acceptable, use the transient keyword for fields that contain direct handles to system resources and contain information relative to an address space. This will prevent deserialization of the class from allowing improper access. It is also advisable to identify all sensitive information as transient.

A developer-defined serializing method for a class should not pass an internal array to any data input/data output method that takes an array, because all data input/data output methods can be overridden.

- *Make classes undeserializable:* Classes, whether serializable or not, should be explicitly made deserializable to prevent an attacker from creating a sequence of bytes that deserializes to an

instance of a given class with values of the attacker's choosing — (that is, allowing the attacker to choose the object's state. Add the following method to all classes:

```
private final void
readObject(ObjectInputStream in)
    throws java.io.IOException
{
    throw new java.io.IOException("Class
cannot be deserialized");
}
```

- *Do not compare classes by name.* This will prevent attackers from defining classes with identical names, which are then granted the privileges of the valid classes with the same names. To determine whether an object has a given class do not use a construction such as this:

```
if (obj.getClass().getName().equals("Foo")) {
```

Instead, use `getClass()` to declare two classes, and then use the `==` operator to compare them, as follows:

```
if (a.getClass() == b.getClass()) {
```

If the program must determine whether an object has a given classname, use the current namespace (of the current class's `ClassLoader`) to determine that name, such as the following:

```
if (obj.getClass() ==
this.getClassLoader().loadClass("Foo")) {
```

Rule of thumb: Avoid comparing class values. Instead, design class methods and interfaces so that such comparisons are not required.

- *Use `System.*` cautiously.* Be very careful, when writing Java servlets and JSPs, of how the `System.*` command is used, *especially* `System.Runtime`.

You may also be interested in applying structured programming techniques to Java development. There is an interesting white paper about this at <http://www.ulst.ac.uk/cticomp/gibbons.html>.

D.3.1 PRIVILEGED JAVA CODE

The Java platform access control mechanism protects system resources from unauthorized access by making sure that the calling code has the appropriate permissions for accessing that resource. When a resource access is attempted, all code traversed by the execution thread up to that point must have appropriate permissions to access the resource.

However, some code may need to call system services that access resources to which the calling code does not have appropriate access permissions. For example, client code with the *RuntimePermission* to load a certain native library may call the *loadLibrary* system service; *loadLibrary* requires read access to the native library file. However, the client code does not have the appropriate *FilePermission* to allow it to access the library file. A check on this execution thread for *FilePermission* would cause the operation to fail.

The solution to this problem is to use the API for privileged blocks, which allows the developer to mark a called code block as privileged, so that code block can call services based on its own permissions, even though the code calling that block does not have those permissions. The privileged block API acts in essence as a wrapper for the code block that is granted privilege. (See <http://java.sun.com/j2se/sdk/1.2/docs/guide/security/doprivileged.html> for a detailed description of the API for privileged blocks).

Use privileged code blocks sparingly, if at all. There are certain privileged services that can be performed by system code on behalf of unauthorized clients even if the code is not encapsulated within a privileged block. When writing privileged Java code, take these measures:

1. Write the code block to be as short and simple as possible. Enable privileges for as little code as possible so that the privileged code can be easily audited to ensure that it accesses only the minimal amount of protected resources for a minimal amount of time (i.e., least privilege).
2. Beware of public methods or nonpublic methods, or both, invoked by public methods that wrap privileged blocks dealing with tainted variables—variables set by the caller by being passed as parameters. Such variables will not be controlled by the privileged code. When a method is public, anyone can call it. If a public method is in a protection domain that allows access to all system properties (e.g., if it is on the boot-classpath), it will grant access to all system properties to any piece of code (including applets) that calls it. Even if the method invokes a protected function within a public class, anyone could extend the class and call the protected function (unless the class is in a restricted package; more information follows). If a privileged block is wrapped by a method that accepts a tainted parameter or variable, the tainted variable could compromise the security of such a code. Methods that accept caller-set variables and parameters should always be private methods that cannot be called from outside their own class.
3. For a code to perform a task that untrusted code or applets would not normally be permitted to perform, the code must be wrapped in a privileged block, which will perform the task on behalf of the untrusted code or applet. Tasks that cannot be performed by untrusted code include these:
 - Reading system properties
 - Reading files (even files in `java.home`)

- Opening sockets
- Writing files (e.g., saving properties in appletviewer)
- Loading dynamic libraries with `System.loadLibrary` or `Runtime.getRuntime.loadLibrary`.

Refer to the Java programmer's manual for specific guidance on use of the *doPrivileged* API.

Note that the `doPrivileged()` method can be invoked reflectively using `java.lang.reflect.Method.invoke()`, in which case the privileges granted in privileged mode are not those belonging to `Method.invoke()`, but those belonging to the nonreflective code that invoked it. Otherwise, system privileges might be erroneously (or maliciously) conferred on user code. The same is true when using the existing API via reflection.

Recommended books and on-line resources for Java security appear in Appendix B.

D.4 HYPERTEXT MARKUP LANGUAGE

Allowing any hypertext markup language (HTML) in user input is a risk. If possible, HTML tags should be strictly prohibited from user input. If this is not possible—such as in a Web mail application or message board—implement an extremely restrictive list of the few HTML tags that users may include in their input.

The following HTML tags can cause security problems because they open access to external pathnames to invoke external logic, reformat the Web page presentation, or load a large object such as a graphic file. For this reason, these tags should be prohibited in user input:

```
<APPLET>
<BODY>
<EMBED>
<FRAME>
<FRAMESET>
<ILAYER>
<IMG>
<LAYER>
<META>
<OBJECT>
<SCRIPT>
<STYLE>
```

In addition, the following attributes should be used with caution:

```
STYLE
SRC (e.g., <IMG SRC=" ">
HREF (e.g., <A HREF=" ">
TYPE
```

As for the HTML that you post as Web content, we strongly recommend running all HTML through an HTML validator before posting it. There are several on-line validators. We have found the Web Design Group HTML validator the most useful, because unlike many of the others, it gives warnings for valid but dangerous HTML constructions. They include unclosed start-tags (e.g., `<p><img src=gov alt=bar</p>`), unclosed end-tags (e.g., `<p>text</em</p>`), empty start and end-tags (e.g., `<p>text</>`), and net-enabling start-tags (e.g., ``). We have also found the errors detected by the HTML Web Design Group validator to be the most relevant. For example, it does not report XML errors when you only want to validate HTML; by contrast, the on-line W3C HTML validation service reports both XML and HTML errors during the same validation. You can find the validator at <http://www.htmlhelp.com/tools/validator/>.

The introduction of scripting languages and interactive capabilities in HTML 4.0 introduced a number of security risks associated with the automatic execution of programs written by the sender but interpreted by the recipient. Browsers that execute such scripts or programs must be extremely careful to ensure that untrusted software is executed in a protected environment.

D.5 XML AND SDML

D.5.1 EXTENSIBLE MARKUP LANGUAGE

The emerging extensible markup language (XML) security standards define XML vocabularies and processing rules to meet security requirements. These standards use legacy cryptographic and security technologies, as well as emerging XML technologies, to provide a flexible, extensible and practical solution toward meeting security requirements. The emerging XML security standards include

- XML digital signature (DSig) for integrity and signing solutions
- XML encryption for confidentiality
- XML key management (XKMS) for public key registration, location, and validation
- Security assertion markup language (SAML) for conveying authentication, authorization, and attribute assertions
- XML access control markup language (XACML) for defining access control rules
- Platform for privacy preferences (P3P) for defining privacy policies and preferences.

XML security defines a common framework and processing rules that can be shared across applications using common tools, avoiding the need for extensive customization of applications to add security. XML security reuses the concepts, algorithms, and core technologies of legacy security systems while introducing changes necessary to support extensible integration with XML. This allows interoperability with a wide range of existing infrastructures and across deployments.

As noted, the XML security standards define XML vocabularies for representing security information, using XML technologies, such as XML schema, for definition. An example is the <KeyInfo> element defined in the XML Dsig recommendation for carrying signing or encryption key information. This definition is used in a number of the specifications. The specifications define a shared meaning for the XML vocabularies.

For an extensive discussion of the various XML security features and their use, see <http://home.earthlink.net/~fjhirsch/xml/xmlsec/starting-xml-security.html>.

D.5.2 SIGNED DOCUMENT MARKUP LANGUAGE

Until the XML DSIg (along with other XML security standards) is finalized, a possible interim alternative to create digitally signed tagged text documents is signed document markup language (SDML), developed by the Financial Services Technology Consortium (FSTC) as part of the Electronic Check Project. SDML enables cryptographic signatures to be embedded in structured, tagged-text documents. Specifically, SDML is designed to

- Tag the individual text items making up a document
- Group the text items into document parts that can have business meaning and can be signed individually or together
- Allow document parts to be added and deleted without invalidating previous signatures
- Allow signing, cosigning, endorsing, coendorsing, and witnessing operations on documents and document parts.

SDML documents are hashed and cryptographically signed using public key signature algorithms. The signatures become part of the SDML document and can be verified by subsequent recipients as the document travels through the business process. SDML does not define encryption, because encryption is between each sender and receiver in the business process and can differ for each link depending on the transport used.

For more information on SDML, see <http://www.w3.org/TR/NOTE-SDML/> and <http://www.xml.org/xml/zapthink/std403.html>.

D.6 ASP AND JSP

D.6.1 DO NOT STORE SENSITIVE DATA IN THE ASP OR JSP PAGE

The sensitive data that developers are most likely to want to store in JSP pages are usernames and password combinations for accessing various resources (e.g., membership directories, database connection strings). Authentication credentials can be entered manually, by the user, or automatically by various wizards or design time controls. Although *SP scripts are processed on the server, and only the

results are sent to the client, a number of known security vulnerabilities in Web servers have allowed the source of *SP pages to be displayed by the browser rather than executed.

For example, two very well-known bugs in the Microsoft Internet Information Server (IIS) Web server have repeatedly caused ASP to be displayed when the user appended to the end of the URL pointing to the ASP file a dot (.) or the string ::\$DATA – as in the following:

```
http://<site>/anypage.asp.  
http://<site>/anypage.asp::$DATA
```

Another anomaly, known as the Translate:f bug, has also been seen to allow the same outcome on IIS servers. Similarly, two recent anomalies have affected BEA Weblogic servers and IBM WebSphere servers. These are documented at the following:

<http://www.foundstone.com/FS-072800-9-BEA.txt>

¾ and ¾

<http://www.foundstone.com/FS-072400-6-IBM.txt>

The Allaire JRun JSP engine has had a different anomaly that allows the same outcome reported. This is documented at the following:

<http://www.allaire.com/handlers/index.cfm?ID=16290&Method=Full>

D.6.2 JAVA SERVER PAGES

The Java Server Pages (JSP) technology facilitates the creation and management of dynamic Web content by embedding Java code logic inside HTML and XML documents. The JSP engine will preprocess and convert the JSP into a Java Servlet (servlets). Subsequent requests for the page will result in the Web server responding with the output generating the corresponding servlet. Although they are functionally the same, JSP represents a reversed approach to dynamic content generation compared with servlets, in that the focus is on HTML documents with embedded Java code instead of a servlet with embedded HTML tags. JSP offers enhanced performance and session management by using Java threads to handle multiple servlets running inside one process. CGI scripts generally require the creation and destruction of a process for each request. JSP differs greatly in architecture from most server-side technologies in that it is an object-oriented, component-based architecture, and it is just one of the APIs that a developer could use under the Java 2 Enterprise Edition (J2EE) platform.

By the nature of providing access to resources on a server, insecure Java servlets generated from JSP pages can put at risk any or all of the server, the network on which the server resides, the clients accessing the pages, and, through worm distribution attacks, the entire Internet. Despite all of the inherit security features of Java, it does not make applications secure on its own. It is not difficult to write insecure Java, especially when developing JSP and servlets. Validating input and controlling access to resources always need to be considered. Furthermore, JSP is a complex architecture, where in many

components come together; as a result, the interactions between them are often sources of security breaches.

Discussions follow on specific security concerns to be aware of, as well as countermeasures to implement when using JSP.

D.6.2.1 Input from Untrusted Users

Input from an untrusted user originates from the client (browser) but can reach the server through many different methods, and sometimes under disguise. Some common sources of user input for a JSP page are:

- The parameter string portion of the URL
- Data submitted by HTML forms through *POST* or *GET* requests
- Data temporarily stored in the client browser (cookies)
- Queries to a database.

The problem with user input is that it can be interpreted by the server-side applications and thus an attacker can craft the incoming data so as to control some vulnerable aspect as of the server. These vulnerabilities can present them as points of access to data identified by user supplied qualifiers.

JSP can access native code libraries through a Java Native Interface (JNI) and execute external commands. Every Java application has a single instance of class run time that allows the application to interface with the environment in which the application is running. The class run time has an `exec()` method, which interprets its first arguments as a command line to execute in a separate process. If parts of the command-line string must be derived from user input, this input must be filtered to ensure that only the intended commands are executed, with the intended arguments. It is also possible for an attacker to modify environment variables in the server environment and affect the execution of external commands, for example, by changing a *PATH* variable to point to malicious code. To avoid this risk, set the environment explicitly before making external calls. This can be done by using `exec(String cmd, String[] envp)` method signature and providing an array of environment variables as the second argument. The variable must have the format `name=value`.

A similar situation arises when user input is used to identify input or output streams that the program opens. Access to files, databases, or other network connections must not depend on invalid user input. The following JSP construct accessing the Java Database Connectivity (JDBC) API service (an API used for database operations) is highly insecure. The attacker can embed command separation characters in the submitted input and execute unwanted commands on the SQL server; this is also known as Uniform Resource Locator (URL) manipulation.

```
<%@ page import="java.sql.*" %>
<!-- open a database connection here -->
```

```
<%  
    Statement s = connection.createStatement();  
    String dbQuery = "SELECT * FROM USER_DATA WHERE  
        USER = " +  
        request.getParameter("username");  
    ResultSet result = s.executeQuery(dbQuery);  
%>
```

If the username contains a semicolon, as in

```
http://server/db.jsp?  
username=john;SELECT%20*%20FROM%20SYSTEM_RECORDS
```

Then the attackers could gain access to or damage parts of the database that they are not entitled. Some SQL servers will ignore the whole query, but others will proceed and execute the two commands. The solution to this problem is achieved through appropriate input validation.

D.6.1.2 Input Validation

Input validation consists of performing syntactic and sometimes semantic checks on data derived from external (un trusted) sources. Assume that all users are untrusted and never assume they will input valid data. Depending on the criticality of the application, the actions performed as a result of input validation may be one or more of the following:

- Escaping unsafe syntactic elements
- Replacing syntactic elements with safe elements
- Reporting error conditions
- Strong type checking
- Checking the length of string arguments received from a URL query.

Design applications that allow only validated user input. The following section details two methods of providing data input validation using JSP.

D.6.1.2.1 Client-Side JSP Validation Scripts

Although JavaScript can be embedded in the JSP in, or called externally by, the client (browser) to perform validation of the length and type of input side that checks, this technique is not an acceptable means of data input validation. Users can disable and bypass scripting performed in the browser. Client-side scripting should be used only in combination with server-side validation, to alleviate tasks performed by the server.

Client-side scripting should be used only if timing constraints in the application make the delay imposed by server-side validation unacceptable. In this case (and only this case), input validation may be distributed between client and server, using a carefully written client-side script to perform the initial validation of data, and then forward that data to the server for a confirmation check of its validity by the server. Implemented correctly, this approach should result in the client filtering out obviously problematic information and never forwarding it to the server. If client-side validation is performed correctly, the server will receive only data that are acceptable in its business logic and, as a result, should be able to execute fewer validations.

The following script checks for invalid user input by checking for bad character data:

```
<SCRIPT LANGUAGE="JavaScript">
function SYMBOL_CHECK(TheObjValue)
{
    if (TheObjValue.match(/[\\"\\/:*?<>|]/)) {
        return 1;
    }
    return 0;
}
function SYMBOL_CHECK2(TheObjValue){
if (TheObjValue.match(/[\\"\\/:*?<>|]/)) {
    return 1;
}
    return 0;
}
function SYMBOL_CHECK3(TheObjValue){
if (TheObjValue.match(/[^a-zA-Z0-9]/)) {
    return 1;
}
    return 0;
}
}
```

This script can be used by another script that checks for null values and proper length of username input.

```
function checkForm()
{
    var error_string = "";

    // check the firstname field
    if (window.document.the_form.firstname.value ==
    "")
    {
        error_string += "First name missing.\n";
    }
    // check the lastname field
```

```
    if (window.document.the_form.lastname.value ==
"")
    {
        error_string += "Last name missing.\n";
    }
    // check the username field
    if (window.document.the_form.username.value ==
"")
    {
        error_string += "User name is missing.\n";
    }

    if
(window.document.the_form.username.value.length
    < 8)
    {
        error_string += "Username must be between 8
and
            12 chars";
    }

    if
(window.document.the_form.username.value.length
    > 12)
    {
        error_string += "Username must be between 8
and
            12 chars";
    }
}
// check for bad character data
var username =
    window.document.the_form.username.value;
var result = SYMBOL_CHECK(username);
if (result == 1)
{
    error_string += "Bad Chars";
}

if (error_string == "")
{
    return true;
}else {
    error_string = "The following omissions
where
        found in the form: \n" + error_string;
    alert(error_string);
    return false;
}
```

```
}  
</SCRIPT>
```

Here is an HTML snippet of a form that processes the data on the client before handling input over to a servlet. The `onSubmit` function of the form will initialize the JavaScript function `checkForm()`:

```
<FORM NAME = "the_form" ACTION="someservlet"  
METHOD="POST"  
  onSubmit="var the_result = checkForm();  
            return the_result;">  
  First Name: <INPUT TYPE="text" NAME="firstname">  
  Last Name: <INPUT TYPE="text" NAME="lastname">  
  User Name: <INPUT TYPE="text" NAME="username">  
  <INPUT TYPE="submit" value="Submit New User">  
  <INPUT TYPE="reset" />  
</FORM>
```

Notice that the HTTP method for a server request is *POST*. Using *POST* will not display the query string in the URL. This is very important when sending username and password attributes across a network. If you do not explicitly assign a variable to the *METHOD* attribute, the default will be *GET*. Again, this is only a snippet of what can be done to ensure the application will accept only valid data. Once again: Do not rely on client-side scripting as the main protector against invalid data.

D.6.1.2.2 Server-Side JSP Validation

Server-side validation gives you complete control over how data are validated on the server. Server-side validation can be performed by a JSP script or servlet. That script or servlet receives the form action submitted by the client and verifies that the data are within acceptable limits. Depending on the outcome of the validation, the script or servlet:

- Accepts the data or
- Rejects the data and returns an error message to the user and
- Redisplays the form to the user with a description of the correction required, and asks the user to reenter the data correctly and resubmit the form.

The only drawback of server-side validation is that the need for the user to resubmit requests for validation each time can incur a delay between the submission of an update by the client, as well as a delay in the acceptance of that update after validation processing by the server. If this delay is unacceptable in a particular application, it may be possible to judiciously distribute a portion of validation to the client, as described in the forgoing “Client-Side JSP Validation Scripts”. Combined with client-side scripting, the examples in the following sections should greatly help in your development of secure JSPs and servlets.

Below is an example of how server-side validation can be done using a JSP form, a servlet for validation, and JavaBean to encapsulate user input data. The form will collect information from the JSP form and validate it through the servlet, which serves as a form handler. The fields in the form will be simple: Last Name, First Name, E-mail, and Social Security number. This example will illustrate basic validation on data. We want to make sure that the user enters his or her first and last name, that the e-mail is actually an e-mail address, and that the Social Security number has the correct number of digits—9. If an error occurs, the user will be sent back to the form to try again.

D.6.1.2.3 Using JavaBean to Handle Invalid Input Submissions

Using a JavaBean makes it easier to repopulate the form fields with the user's data after an invalid submission. Because another servlet performs the input validation, you need to specify only the GET and SET methods. The bean tags will update the form after it has been populated, as in the following example:

```
public class UserBean {
    private String firstName;
    private String lastName;
    private String ssn;
    private String email;

    //default constructor
    public UserBean() {
        this.firstName = "";
        this.lastName = "";
        this.ssn = "";
        this.email = "";
    }
    //get methods
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public String getSSN() {
        return ssn;
    }
    public String getEmail() {
        return email;
    }
    //set methods
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

```

    }
    public void setSSN(String ssn) {
        this.ssn = ssn;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String toString() {
        String userData = "";
        userData = firstName + " " + lastName
            + "\n" + ssn + "\n" + email;
        return userData;
    }
}

```

D.6.1.2.4 Security-Aware JSP Forms

The JSP form that follows will populate the `UserBean` with any data that are present in the request. If there is a validation problem following submission, the request will be directed back to the page populating the form with data. This is handled by the `FormHandlerServlet` in the example. Whether or not there are empty data, use the `<jsp:getProperty>` tags to populate default values in the form fields.

```

<HTML>
<% /* call the UserBean class */ %>
<jsp:useBean id="form" class="UserBean">
    <jsp:setProperty name="userbean" property="*" />
</jsp:useBean>

<BODY>
<%
    String[] errors =
    (String[])request.getAttribute("errors");
    if (errors != null && errors.length > 0) {
%>
<B>Please correct the following errors</B>
<UL>
<% for (int i=0; i < errors.length; i++) { %>
    <LI><%= errors[i] %>
<% } %>
</UL>
<% } %>

<FORM METHOD=POST ACTION="FormHandlerServlet"
    method="post">
<INPUT TYPE="text" NAME="lastname"
value="<jsp:getProperty

```

```
        name='form' property='lastname' />">
<B>Last Name</B><BR>
<INPUT TYPE="text" NAME="firstname"
value="<jsp:getProperty
        name='form' property='firstname' />">
<B>First Name</B><BR>
<INPUT TYPE="text" NAME="ssn"
value="<jsp:getProperty
        name='form' property='ssn' />">
<B>Social Security #</B> (123456789)<BR>
<INPUT TYPE="text" NAME="email"
value="<jsp:getProperty
        name='form' property='email' />">
<B>E-mail #</B> (user@host)<BR>
<INPUT TYPE="submit" Value="Submit Form">
</FORM>
</BODY>
</HTML>
```

The following servlet will perform the necessary form handling, including validating user input, performing whatever operation is required by the application, and redirecting the user to the next page in the process. This is the crux of all input validation: it enables you to exercise full control over data checking. In addition, the servlet does not have to be modified as the form changes, which increases reusability.

STEP 1: The first step performed by the servlet is to retrieve the values of the request attributes:

```
public class FormHandlerServlet extends HttpServlet
{
    public void service(HttpServletRequest req,
                        HttpServletResponse res)
        throws ServletException, IOException {
        Vector errors = new Vector();
        // get attribute values from request
        String firstname =
        (String)req.getAttribute("firstname");
        String lastname =
        (String)req.getAttribute("lastname");
        String ssn =
        (String)req.getAttribute("ssn");
        String email =
        (String)req.getAttribute("email");
```

STEP 2: The servlet checks for valid input using the specified methods that return flags if the servlet finds an error:

```
// check for valid input
    if (firstname == null) {
        errors.add("Please provide your first
name");
    }
    if (lastname == null) {
        errors.add("Please provide your first
name");
    }
    if (!isValidSSN(ssn)) {
errors.add("Plase specify a valid SSN, i.e
1234556789");
    }
    if (!isValidEmail(email)) {
        errors.add("Email address must contain
a @ symbol");
    }
    private boolean isValidSSN(String ssn) {
        // check for nine characters
        return (ssn.length() == 9 &&
ssn.indexOf("-") == -1);
    }
    private boolean isValidEmail(String email)
{
        // check for "@" somewhere after the
first character
        return (email.indexOf("@") > 0);
    }
}
```

STEP 3: The servlet checks for validation errors. If there are any validation errors, the servlet returns the form to the user and asks him to correct the invalid input. If no validation problems are found, the servlet returns the next page in the process to the user:

```
if (errors.size()==0) {
    // data is ok, dispatch to next page
    gotoPage("next.jsp");
} else {
    // data has errors, resubmit
    String[] errorList =
(String[])errors.toArray(new
String[0]);
    req.setAttribute("errors", errorList);
    gotoPage("form.jsp");
}
```

```
    }
    private void gotoPage(String address,
                          HttpServletRequest request,
                          HttpServletResponse
response)
        throws ServletException, IOException {
        RequestDispatcher dispatcher =

        getServletContext().getRequestDispatcher(address
);
        dispatcher.forward(request, response);
    }
```

D.6.1.2.5 Other JSP Security Issues

As with any other Web technology that uses a CGI protocol, there are other security issues to be aware of, including these:

- *Do not send data via a GET request:* The most trivial method for transferring request data from the client to the server-side application is the GET request. In this method, input data are appended to the URL and are represented in the following form:

```
URL[ ?name=value[ &name=value[ &... ] ] ]
```

This is clearly an unacceptable method to transmit data over the Web. Instead, always use *POST* with an appropriate encryption method, (such as an over an Secure Sockets Layer [SSL] connection).

- *Do not store sensitive user information in cookies:* A security exposure is created when sensitive information is stored in cookies, for two reasons: (1) the whole content of the cookie is visible to the client, and (2) there is nothing to prevent a user from responding with a forged cookie. Together, these vulnerabilities make cookies untrustworthy and thus unacceptable mechanisms for storing security data or other sensitive information.
- *Be aware of implementation vulnerabilities:* Certain versions of JSP implementations have been discovered to contain security vulnerabilities. Before using any JSP product, always refer to the vendor's Web site and download and apply all current bugs and fixes to prevent security vulnerabilities.

D.7 CGI AND PERL

D.7.1 SECURE CGI SCRIPTS

CGI scripts may be quite complex or they may consist of only a few lines of text. The most common scripting languages for Web applications are VBScript, JavaScript, Perl, and Python. Scripts require the existence of a scripting engine before they can run. As a result, scripts are slower than compiled

applications, because they must be interpreted instead of being compiled in advance and running in their native (binary) form.

A script that interacts with a networked client is always a potential target of attack. For that reason, it is vital that CGI scripts be written extremely carefully to ensure that they do not threaten the integrity of the application or gain unauthorized access to the server.

The following recommendations should increase the safety of your CGI scripts:

1. Do not write shell scripts.
2. Turn off server-side *includes* in the directories in which CGI scripts will be stored. Also make sure that the server will not attempt to parse any CGI script for server-side *includes*. Server-side *includes* can be abused by attackers if the directories in which they are available store scripts that directly output data sent to the scripts or that modify HTML. Server-side *include* attacks can be used to
 - Mail the password file (if not shadowed) to the attacker (see example that follows)
 - Mail a map of the file system to the attacker
 - Mail system information from the UNIX */etc* directory to the attacker
 - Start a log-in server on a high port and telnet into the server
 - Cause a denial of service, for example, by initiating a very large file system *find* or another resource-consuming command
 - Erase and/or alter , or do both to the server's log files.
3. Do not allow “<” and “>” in user input: either reject the input, or *escape* the characters (by prepending them with a back slash).
4. Remove all comments from CGI scripts and HTML code.
5. Implement a simple program to validate all HTML input to CGI scripts.
6. Use only well-known libraries to parse CGI input, such as *CGI.pm*, to parse Perl scripts.
7. Do not use the `eval` statement.
8. Never trust the client. Although a well-behaved client will *escape* any characters in a query string that have special meaning to the system shell—thus avoiding having the CGI script misinterpret those characters—a malicious user may intentionally include special characters in query strings to confuse the CGI script and gain unauthorized access to the server.

9. Do not trust user-supplied data even if the script does not invoke a shell.
10. Do not use the `HTTP_REFERER` header, which originates in the untrustworthy browser and not in the server, for authentication in a CGI program.
11. Be careful when using commands that could fork a shell, such as `system()`, `exec()`, or the back tick (```) in Perl; or `popen()` in C and C++. To avoid invoking the shell when using `system()` and `exec()` in Perl, supply more than one argument to the call, as in this example:

```
system('/usr/lib/routine', '-o');
```

In C and C++, use `open()` instead of `popen()` Here is an example:

```
open(FH, '|-') || exec("program", $arg1, $arg2);
```

In Perl, avoid the functions `open()`, `glob`, and the backtick (```). All three call the shell to expand filename wild card characters. Instead of `open()` use `sysopen()`, or, if using Perl 5.6 or later, use `open()` with three parameters (consult the `perlopentut()` *man* page). Instead of back ticks, use the `system()` call or an even safer call.

12. Validate all data and do not accept insecure data or metacharacters or pass them to the shell. Better yet, instead of simply detecting the metacharacters, *escape* them.

NOTE: “\” (back slash), the shell escape character, may itself be present in input data and cause problems. Note also that ASCII 255 is treated as a delimiter by some shells and may need to be escaped.

13. Encode dynamic output to prevent malicious scripts from being passed to the user.
14. In your CGI script, turn on the warning flag (`-w`) to warn of any potentially dangerous or obsolete statements.
15. Consider running the CGI script in a restricted environment, such as the Perl sandbox described in the `perlsec()` *man* page, or the *Safe* module in the standard Perl distribution.
16. Do not make assumptions about the operating environment. Do not write *cgi-bin* under the assumption that it will run in a safe environment on the Web server. It is possible for an attacker to execute a *cgi-bin* program (or force its execution) in an unexpected context. Like all programs, *cgi-bin* programs (particularly those that run `setuid` on UNIX servers) should sanitize their own environments before creating any shells or invoking any other programs. At a minimum, set the value of the *PATH* and *IFS* environment variables to a known state by resetting the environment to *null*, and then building a new, known environment (on UNIX, use `undef()` to reset the environment).

NOTE: In taint mode, Perl will warn you if the script attempts to call `system()` without first setting the `PATH` and `IFS` variables appropriately.

17. Always use Perl taint mode unless there is a really good reason for not doing so. Perl taint mode is one of the best tools for ensuring the security of Perl CGI scripts. Use Perl5 rather than other versions of Perl to be sure the language provides taint mode support.

D.7.2 PERL TAINT MODE

Taint mode causes Perl to perform extra security validations when accessing variables and making function calls. Taint mode checks ensure that any tainted data received from outside the program are not used, directly or indirectly, to modify files, processes, or directories.

Specifically, taint mode prevents data derived from outside the program from accidentally affecting anything else outside by program. It marks such data as tainted. All externally obtained input is marked as tainted, including:

- Command-line arguments
- Environment variables
- Locale information (see `perllocale()`)
- Results of the `readdir` and `readlink` system calls
- Results of the `gecos` field of `getpw*` calls
- All file input.

Tainted data may not be used directly or indirectly in any command that invokes a sub shell, nor in any command that modifies files, directories, or processes. There is one important exception: If the script passes a list of arguments to either `system` or `exec`, the elements of that list will not be checked for taintedness. Be especially careful when using `system` or `exec` while in taint mode. Any data value derived from tainted data becomes tainted also.

To untaint data, extract a substring of the tainted data. Do not simply use “*” as the substring that will defeat the taint mode mechanism altogether. Instead, identify safe patterns to be allowed by the CGI script, and use them to extract good values. Check all extracted values to ensure that they do not contain unsafe characters or exceed safe lengths.

When invoking Perl as a `setuid` program or CGI script, place the `-T` flag at the end of the command line to put Perl into taint mode. For example, to do this:

```
#!/usr/bin/perl -T
```

See the `perlsec() man` page for details.

Perl and `mod_perl` systems enable you to add to script input validation of HTML-*escaped* input data. For example, insert the following line of code before any output to eliminate any input that is not an alphanumeric character or a space:

```
$text =~ s/[^A-Za-z0-9 ]*/ /g;
```

D.7.2.1 Other Methods for HTML-*escaping* Data in Perl

1. Use the `HTML::Entities::encode()` function in the `HTML::Entities` module in the `libwww-perl` CPAN distribution to *escape* HTML characters in input data by encoding them into HTML entity references.
2. When using the `Apache::Registry` script or `mod_perl` handler, use `Apache::Util::escape_html()` in the `Apache::Registry` to encode all HTML input.
3. Rather than typing `Apache::Util::html_encode()` every time you need to validate an input, use the `Apache::TaintRequest` module (see Appendix C) to automate *escaping* of HTML data. The action that is applied to the whole script to HTML-*escape* any data item that is found to be tainted and passes all untainted data to the browser without altering it.

D.7.3 CHANGING TO OWNER'S ACCOUNT TO RUN CGI SCRIPTS

Most HTTP daemons (*httpds*) do not change the CGI script's account (username) to that of the script's owner. Instead, they run the CGI script under the "nobody" account. As a result, these scripts require files to be world writable, or UNIX CGIs to be *suid*.

Individual files should never be world writable, even if the directory that contains them is. Making the directory only world writable enables you to have the CGI script to write a file owned by "nobody," and then to restore directory permission afterward (within the limitations of file system disk quotas, etc.)

CGIwrap (see Appendix C) is a good tool for changing change a CGI script's username to run under the owner's account, instead of as "nobody."

Do not use `suidperl`. Instead use `sudo` (see <http://www.courtesan.com/sudo/> for more information).

D.8 STRUCTURED QUERY LANGUAGE

Within Structured Query Language (SQL) itself, there are no known commands or constructs that can, on their own, cause a system to crash or misbehave. However, the *delete* command can be used to maliciously delete data.

Because an SQL statement can be made only by an entity that has been authenticated against a valid database account, SQL statements from users unknown to the database will not be processed.

A number of vulnerabilities exist within several Oracle 9i products and supports services that may cause buffer overflows, denial of service, mismanagement of permissions, or unauthorized data disclosure.

Information on avoiding SQL injection and other database security issues appears in Section 4.5.2.9.

D.9 SHELL SCRIPTING LANGUAGES

Do not write or use system shell or command shell script languages in DoD Web applications.

D.10 TOOL COMMAND LANGUAGE

Tool command language (TCL) comprises two parts: the language and the library. The language is a simple text language used to issue commands to interactive programs. It includes basic programming capabilities. The TCL library can be embedded in application programs. In their effort to make TCL as small and simple as possible, however, its designers have created a language that is somewhat limited from a security standpoint.

Because TCL was designed to be a scripting language, it has few of the capabilities of a full-blown programming language. It has no arrays or any structures from which you create linked lists. It simulates numbers, which slows TCL programs down. As a result, TCL is suitable only for small, simple programs.

In TCL, there is only one data type, *string*. This and its other limitations make programming anything other than very simple scripts in TCL difficult. In addition, TCL executables tend to run slowly. With TCL, developers can accidentally create programs that are susceptible to malicious input strings. For example, an attacker can send characters with special meanings to TCL such as embedded spaces, double quotes, curly braces, dollar signs, and brackets. An attacker might even send input that causes these characters to be created during processing, to trigger unexpected and even dangerous behavior in the TCL program.

For all these reasons, TCL—should not be used to write programs that perform security functions, such as mediating a security boundary.

There is a promising alternative to generic TCL—Safe-TCL, which creates a sandbox in which the TCL program operates. Safe-TCL should be implemented in conjunction with Safe-TK, which implements a sandboxed portable GUI for Safe-TCL. Because it contains its own sandbox feature, Safe-TCL may be a good language in which to implement simple mobile code constructs. See Appendix C for more information.

D.11 PHP

Older PHP versions (4.1.0 and earlier) are less secure than most languages due to the way PHP loads data into its namespace. That is, all environment variables and values sent to PHP over the Web as global variables are automatically loaded into the same namespace with normal variables, enabling

attackers to set these variables to arbitrary values that endure until they are explicitly reset by a PHP program.

When a variable is first requested, PHP automatically creates that variable with a default value. Therefore, PHP programs often do not initialize variables. If the programmer forgets to set a variable, PHP must be explicitly configured to report the problem; by default, it will simply ignore the problem. Thus, by default, PHP allows the attacker to completely control the values of all variables in the program, unless the programmer has written the program to explicitly reset all PHP default variables as soon as it starts executing. Failing to reset a single variable may create a vulnerability in the PHP program.

For example, the following PHP program is intended to implement an authorization check to ensure that only users who submit the correct password are allowed access to the sensitive information. However, simply by setting *auth* in his Web browser, the attacker can subvert this authorization check:

```
<?php
if ($pass == "hello")
    $auth = 1;
...
if ($auth == 1)
    echo "sensitive information";
?>
```

It is possible to disable this vulnerability, and eliminate the most common PHP attacks, by setting `register_globals` to *off* (the default in older versions of PHP is *on*, and these versions are hard to use with `register_globals` set to *off*).

In PHP version 4.2.0 and later, the default for `register_globals` is *on*. This is one of several reasons to use only later PHP versions. In PHP 4.2.0, external variables received from the environment, the HTTP request, cookies, or the Web server, are no longer registered in the *global scope* by default. Instead, they are accessed by using the language's new *Superglobal* arrays. Several other special arrays—most notably `$_REQUEST`—make it easier to develop PHP programs when `register_globals` is *off*.

Note that many third-party PHP applications will not operate correctly (or in some cases at all) with `register_globals` set to *off*. Therefore, it may be possible to set this selectively only for the programs that can operate. For example, on an Apache Web server, insert the following lines into the file *.htaccess* in the PHP directory:

```
php_flag register_globals Off
php_flag track_vars On
```

Also consider using directory directives to further control it. Further, note that the *.htaccess* file itself will be ignored unless the Apache Web server has been configured to permit overrides. Check to be sure that the Apache global configuration is not configured with `AllowOverride` set to *None*. Instead, it must “AllowOverride Options” in its configuration file. This will enable you to write helper functions that simplify loading the data needed by your PHP programs—and only those data.

The later versions of PHP also provide functions that make it easier to specify and limit the input the program should accept from external sources. Routines can be placed in the PHP library to enable users to list the input variables they want to accept, and functions can be written to check the validity of the patterns and types of variables before coercing the program to use them.

There have also been reports of a format string problem in the PHP error reporting library. That problem is expected to be fixed in later a version. However, PHP generally has not had wide enough use to expose other possible security vulnerabilities in the language.

If you cannot possibly set `register_globals` to *off*, write the program to accept only values not provided by the user, and do not trust PHP default values. Do not trust any variable that has not been explicitly set. Remember that these variables must be set for every entry point into the program itself and in every HTML file that uses the program.

The Following guidelines will be useful:

- Begin each PHP program by resetting all variables, including globally referenced variables referenced transitively in included files and libraries, even if this means simply resetting them to the default values. This will entail having to learn and understand all of the global variables that might be used by the functions called in your program. Search through the *HTTP_GET_VARS*, *HTTP_POST_VARS*, *HTTP_COOKIE_VARS*, and *HTTP_POST_FILES* to determine the originator of the data (it should not be a user). This search needs to be repeated whenever a new version of PHP comes out, because it may add a new data source.
- Write the program to record all errors by piping them to error reports in a log file.
- Filter all user information used to create filenames to prevent remote file access. PHP defaults to remote files functionality enabling it to use commands like `fopen ()`, which in other languages can open only a local file, to invoke Web or FTP requests from other sites.
- Use only the *HTTP_POST_FILES* array and related functions to upload files and access uploaded files. PHP allows attackers to temporarily upload files with arbitrary content. This cannot be prevented in the language itself.
- Place only protected entry points in the document tree, and place all other code—that is the majority of code—outside the document tree. Do not place *.inc (include)* files inside the document tree.
- Do not use PHP’s session mechanism, for it still contains security vulnerabilities.

- After validating all inputs, use type casting to coerce nonstring data into the type it should have. Develop helper functions to check and import a selected list of expected inputs. PHP is loosely typed, which can cause problems. For example, if an input has the value 000, it will not be interpreted as 0 or `empty()`. This can present a problem when using associative arrays in which the indexes are strings, meaning that `$data["000"]` is not the same as `$data["0"]`. To make sure `$bar` is assigned the type *double* (after verifying that its format is legal for a double), write `$bar = (double) $bar;`
- Be very careful when using the following functions:

- **Code execution**

```
require()  
include()  
eval()  
preg_replace() (with or without the /e flag)
```

- **Command execution**

```
exec()  
passthru()  
` (backtick)  
system()  
popen()
```

- **Open file commands**

```
fopen()  
readfile()  
file()
```

The foregoing list is not exhaustive; refer to the PHP specification for all code and command execution and file opening commands, and use those commands very carefully. Other rules of thumb when using PHP are these:

- Use `magic_quotes_gpc()` where appropriate. This will help eliminate many kinds of attacks.
- Avoid file uploads. Modify the `php.ini` file to disable them (`file_uploads = Off`).

Refer also to Appendix B, Section B.2.2.4.6.

D.12 PYTHON

As with other languages, be very careful when using functions and calls that allow data to be executed as parts of a program, including these:

Functions

```
exec()  
eval()  
execfile()
```

Calls

```
compile()  
input()
```

Privileged (trusted) Python programs that can be invoked by unprivileged users must not import the `user` module, which causes the `pythonrc.py` file to be read and executed. Doing so could allow an unprivileged attacker to force the trusted program to run arbitrary code.

Python does very little compile-time checking. It implements no static typing or compile-time type checking. Nor does it check that the number of parameters passed is legal for a given function or method. There is an open source program, PyChecker, which can be used to check for common bugs in Python source code. See Appendix C.

Python does support restricted execution through its `RExec` class. `RExec` is primarily intended for executing applets and mobile code, but it can also be used to limit privilege in a program when the code has not originated external to the program. By default, a restricted execution environment permits reading (but not writing) of files, and it does not allow operations for network access or graphical user interface (GUI) interaction. If you are changing these defaults, beware of creating security vulnerabilities in the restricted environment.

Python's implementation calls many hidden methods and passes most objects by reference. When inserting a reference to a mutable value into a restricted program's environment, first copy that mutable value or use the `Bastion` module. Otherwise, the program may change the object in a way that is visible outside the restricted environment.

APPENDIX E: SECURITY-ENHANCING LEGACY APPLICATIONS

There is no clear-cut approach to providing security services to a legacy application. Many non-Web legacy applications support various methods for user access and authentication. Most legacy applications require a client application to be installed on the user's machine that connects to a server over a network. Users are most likely authenticated to the application through a basic username and password combination, which calls a built-in access control list (ACL). This ACL most likely does not include a callable API, which presents a dilemma when security needs change and/or more granular security is required, or both. Techniques presented in this appendix can be used to increase the security of legacy as well as new applications.

E.1 WEB ENABLING FOR SECURITY

Web enabling adds the inherent security features of the Web architecture, such as SSL and client certificate authentication, to legacy applications. It also provides administrators of those applications with centralized management and more control over who is able to use the application. Web enabling, commonly referred to as a portal solution, entails adding the ability for users to be administered through digital rights management software which maps them to the Web-enabled legacy application as well as to new Web applications.

There are a number of COTS products that developers can use to Web enable and add centralized management to legacy applications (see Appendix C). These systems provide a way for a client/server application to communicate via HTTP (e.g., using Microsoft Remote Desktop Protocol (RDP) or Terminal Server, or UNIX X Protocol Engines).

The implementation and architecture of these systems are nontrivial. If you intend to Web enable an application using a COTS tool, be sure to get training in the use of that product and thoroughly read the product's technical documentation.

E.1.1 SECURITY CHALLENGES OF WEB ENABLING

The main challenge when Web enabling a legacy application is integrating the application's existing authentication mechanism with the public key-enabled Web authentication capability. Most legacy applications do not provide an API that allows the developer to override the existing log in module that interacts with a built-in ACL. Consider the following scenario:

1. An SSL session is established between browser and Web portal application.
2. The user submits a certificate to authenticate him to the Web portal.
3. The user clicks on an available hyperlink on the portal, and his session is redirected to a Web-enabled legacy application.

4. Because the legacy application provides no API to override its existing log-in module, the user must reauthenticate to the legacy application by submitting his username and password.

Despite the cryptographic security of the Web portal, once the user reaches the legacy username and password authentication dialogue behind that portal, he can still exploit the inherent weaknesses of static passwords. The fact that he had to be cryptographically authenticated to the Web portal may provide a certain degree of psychological comfort. But that comfort does not translate into actual user trustworthiness of the user once he has made it past the Web portal.

Two possible ways to harmonize the legacy application's authentication with the Web portal's PKI-based authentication are these:

1. Incorporating the legacy application into an overall SSO framework that can map the user's PKI certificate to his legacy username and password in the user's entry in the SSO system's credentials directory.
2. PK-enabling the legacy application to be able to retrieve the user's certificate from the same directory used by the Web portal and to validate that certificate.

E.1.1.1 PK-Enabling of Web-Enabled Legacy Applications

Once a legacy application has been Web enabled, you can implement some of the PKI-based security services of the Web architecture into the application. For example, the portal server can be enabled by SSL/TLS (that is, transport layer security) to request a DoD PKI server session certificate from a trusted certificate authority. The server can then create the SSL/TLS HTTPS session channel for basic user authentication and provide client assurance of a trusted path to the server. Resources on the portal could be configured to require users to authenticate themselves using DoD PKI identity certificates issued by a CA.

An additional benefit of PK-enabling is the ability to add validation of certificate revocation lists (CRLs) to the server. The server can then determine whether the certificate it receives from the user appears on the list of revoked certificates generated by the CA, and to prevent the session from being established if the certificate has been revoked (while issuing an error message to the user).

E.1.2 UPDATING SECURITY IN LEGACY WEB APPLICATIONS

Of course, if the legacy application is itself a Web application, Web enabling will not be necessary. The legacy Web application will most likely fit into an *n*-tier Web application architecture, all of which is predicated on cryptographic security. You could modify the legacy Web application's login page to accept HTTP header variables and then write some code to automatically map header variables to a user profile. This approach works with most Web technologies, including ColdFusion, Active Server Pages (ASP), Java Server Pages (JSP), and HTML.

A three-tiered architecture can be applied to the legacy Web application with a client layer providing the user's view of the application—including client browser and user certificate. Tier 2, the logic layer that the client calls on to access the resource, consists of two components: the middleware (portal or wrapper) and the legacy application server. The middleware is used to control access to the application server, and the application server itself is used to host the legacy Web application. Tier 3 contains the database that is required by the application.

Depending on the network configuration, server communication behind the firewall may need to not be SSL encrypted, since the middleware is acting as a firewall.

E.1.2.1 Reimplementing Authentication in Legacy Web Applications

The example that follows illustrates a rewritten *.jsp* login page that will accept HTTP parameter data from a Web portal server. This example could be applied to similar Web technologies using the required syntax. For this example, assume that the `doLogin()` method is an instance method of a JavaBean Web component or a Java servlet; `doLogin()` performs the database calls to the legacy Web application and performs the mapping between the received parameter and username and password.

```
<% String portalHeader =
request.getHeader("HTTP_PORTAL_USER_PARAM");
if (portalHeader !=null && portalHeader.length() <
12)
{
    doLogin(portalHeader);
}

%>

// Pseudo code for mapping portal data to database

doLogin(String httpParam) {

    if (httpParam != null) {

        // Get database connection
        // Map user info sent from portal to the database
        // Close connection
    }

    // return some flag

    return;
}
```

An extra table or column reflecting the accepted parameter would have to be created in the database so it could be mapped to the username and password fields of the user.

E.1.3 INTEGRATING MAINFRAMES INTO WEB APPLICATIONS

Security for the mainframe was not designed with the Internet in mind. Rather, mainframe security was designed for a closed, well-defined, and tightly controlled environment. Key characteristics of such environments include a known and relatively trusted user population, a well-defined set of applications, and firm connectivity boundaries. Application developers need to leverage existing backend mainframe security systems through such mechanisms as mapping digital certificates to RACF (the IBM mainframe security application).

In addition, when integrating legacy systems as backend servers in Web applications, developers must make sure that security does not stop with the connection between the Web server and the browser. Instead, the developer should

- *Establish end-to-end user authentication:* Set up an end-to-end user authentication mechanism that reaches from the Web browser to the mainframe or other backend server.
- *Establish an end-to-end encrypted data channel:* Use encryption from the browser to the Web server and from the Web server to the backend system. This is done to ensure that sensitive data are not exposed in transit over external and internal networks.
- *Implement PKI security end-to-end:* Deploy PKI all the way from the browser to the backend system, integrating it with existing security systems, such as RACF.

E.2 APPLICATION FIREWALLS

Application firewalls can be used to minimize incoming code allowed into the legacy application. These firewalls filter undesirable content from user input. They also prevent transmission into the legacy system of undesirable mobile code and other active content. Some application firewall products are listed in Appendix C.

E.3 SECURITY WRAPPERS

Security wrappers are software programs that encapsulate other programs that are suspected of containing bugs or vulnerabilities. The wrapper program intercepts input intended for the wrapped program and performs various integrity checks on that input on behalf of the wrapped program. Unless the input passes these checks, it will not be passed into the wrapped program. Such wrappers are usually programmed to restrict the syntax of input to finite-length strings and safe character sets. An example is a wrapper that would protect a program that would otherwise accept data strings that could induce buffer overflows or other security errors. One such available wrapper is *overflow_wrapper.c*, available from AUSCERT (see Appendix C). Following are several ways to implement wrappers:

- Implement application enhancements and additions in Java, to take advantage of the Java sandbox capability.
- Perform targeted rewrites of application code to reduce size and functionality: Rewrite the application code in a way that reduces its size and complexity. Break the application program up so that separate smaller programs perform different functions, instead of by one large, complex program.
- Implement static type checking at compile or load time (e.g., in Java, the type system is enforced both by the Java compiler and the Java bytecode verifier).
- Implement application enhancements and additions using proof-carrying code, whereby the code contains its own proof that it will not perform certain classes of illegal operations.
- Implement dynamic array bounds checking at run time to ensure that array bounds are respected and programs cannot access array members outside of the bounds of the array. This will help ensure that programs cannot treat arbitrary chunks of memory as members of an array.
- You can use Immunix StackGuard (see Appendix C) with the random canary (rather than the terminator canary) to enhance GNU C Compiler (GCC) functionality by adding an inspection of the process states integrity at compile time. This inspection will ensure that a buffer overflow attack has not corrupted any process state.
- Mark the stack segments of each application process's address space as nonexecutable. Kernel patches are available for both Linux and Solaris.

End