



ADDRESSING CHALLENGES IN APPLICATION SECURITY

An in-depth examination of the importance of addressing web application issues throughout the Software Development Life Cycle (SDLC)

KENNETH GRAF

A whitepaper from Watchfire

TABLE OF CONTENTS

1. Understanding Application Security Challenges.....	1
1.1 Classic Application Security Challenges.....	1
1.2 Application Security Defined.....	2
2. Application Security Threats.....	3
2.1 Impersonation.....	3
2.2 Tampering.....	4
2.3 Repudiation.....	4
2.4 Information Disclosure.....	4
2.5 Denial of Service (DoS).....	5
2.6 Elevation of Privilege.....	5
3. Reviewing Security Concerns for Existing Applications.....	5
3.1 Discovery and Baselines.....	6
3.2 Risk assessment and assignment.....	6
3.3 Shielding and damage control.....	6
3.4 Ongoing monitoring and review.....	7
4. Resolving Errors throughout the SDLC.....	7
4.1 Cost of Fixing Errors in the SDLC.....	7
4.2 Types of Errors in the SDLC Process.....	8
4.3 General Approaches to Application Security Testing.....	8
5. Define Your Approach.....	10
5.1 Security Awareness.....	10
5.2 Application Risk and Liability Categorization.....	11
5.3 Zero Tolerance Enforcement.....	11
5.4 Security Testing Integrated into Development Process.....	11
6. Validate Your Methodology.....	12
6.1 Security Awareness.....	12
6.2 Application Risk and Liability Categorization.....	12
6.3 Zero Tolerance Enforcement.....	12
6.4 Security Testing.....	13
Appendix A: Requirement Phase Security Considerations.....	14
Appendix B: Design Phase Security Considerations.....	16
Appendix C: Coding Phase Security Considerations.....	18
Appendix D: Using CMMI to improve Application Security.....	20
Appendix E: Microsoft's DREAD-based risk scoring.....	21
Appendix F: Event-Driven Security Testing.....	23
7. Further Reading.....	25
About Watchfire.....	26

Copyright © 2005 Watchfire Corporation. All Rights Reserved. Watchfire, WebCPO, WebXM, WebQA, Watchfire Enterprise Solution, WebXACT, Linkbot, Macrobot, Metabot, Bobby, Sanctum, AppScan, the Sanctum Logo, the Bobby Logo and the Flame Logo are trademarks or registered trademarks of Watchfire Corporation. GómezPro is a trademark of Gómez, Inc., used under license. All other products, company names, and logos are trademarks or registered trademarks of their respective owners.

Except as expressly agreed by Watchfire in writing, Watchfire makes no representation about the suitability and/or accuracy of the information published in this whitepaper. In no event shall Watchfire be liable for any direct, indirect, incidental, special or consequential damages, or damages for loss of profits, revenue, data or use, incurred by you or any third party, arising from your access to, or use of, the information published in this whitepaper, for a particular purpose.

1. UNDERSTANDING APPLICATION SECURITY CHALLENGES

Today’s web application attacker can use your own applications to expose, embarrass and steal from you. Firewalls and SSL are commonplace yet, according to recent studies, three out of four websites are vulnerable to attack, and the vast majority of these attacks are application security attacks.¹ Companies rely on network and host security, but often these measures are simply not enough to prevent these web application attacks.

Application security is different for network and host security. The traditional approaches to implement network and host security do not apply at this level. This paper will tell you why, what to do about it, and provide a roadmap to improving your own application security.

1.1 CLASSIC APPLICATION SECURITY CHALLENGES

Companies will face a wide variety of security challenges. The following tables show the business and technical concerns raised by some of these challenges. We will present potential solutions to these problems later in this paper. Watchfire’s approach of using security awareness, application risk assessment, zero tolerance and complete testing may also help to address your application security concerns.

1.1.1 Insurance: A top-brand company with millions of individual and group customers

Business Challenges	<ul style="list-style-type: none"> 🔥 Meet regulatory compliance 🔥 Proactively secure sensitive customer records 🔥 Integrate application security into multi-tiered security strategy supporting over 4000 physicians in 60 hospitals 🔥 Eliminate costs associated with finding and fixing post-production security issues
Technical Challenges	<ul style="list-style-type: none"> 🔥 Site growing rapidly 🔥 Identified numerous application security flaws production in sites (discovered through audit) 🔥 Ensure secure customer information and transactions 🔥 95 percent of data is considered confidential.

1.1.2 Finance: A top U.S. commercial bank with assets over \$200B

Business Challenges	<ul style="list-style-type: none"> 🔥 Meet corporate mandate to build application security into development life cycle 🔥 Over 3000 legacy and new applications 🔥 Reduce overall cost of development life cycle 🔥 Company outsourcing >\$1M a year for “ethical hacking” to detect vulnerabilities prior or after deployment
Technical Challenges	<ul style="list-style-type: none"> 🔥 Massive well-known site 🔥 Application development distributed across many business units 🔥 Inconsistent manual testing and code review by developer 🔥 Developers had no tools or knowledge of security testing techniques.

¹ “All-Out blitz against Web app attacks,” *Network World*, May 17, 2004.
<http://www.networkworld.com/techinsider/2004/0517techinsidermain.html>

1.1.3 Pharmaceutical: A top-tier, research-driven pharmaceutical products company

Business Challenges	<ul style="list-style-type: none"> 🔥 Meeting organizational and governmental regulations for data protection 🔥 Accurately assessing risk associated with each application 🔥 Frugally applying resources to secure applications and data
Technical Challenges	<ul style="list-style-type: none"> 🔥 Numerous disparate development groups and lines of business 🔥 Multiple acquisitions and strategic partnerships 🔥 Highly regulated confidential data 🔥 Adhering to multinational regulations

1.1.4 Entertainment & Media: Top U.S. television network

Business Challenges	<ul style="list-style-type: none"> 🔥 Highly visible brand 🔥 Multiple high profile and often controversial media properties
Technical Challenges	<ul style="list-style-type: none"> 🔥 Very dynamic regularly changing sites 🔥 Completely decentralized application development 🔥 Small application security team 🔥 Time critical production schedules

1.2 APPLICATION SECURITY DEFINED

The Open System Interconnection (OSI) reference model defines seven network protocol layers, and every message goes through all seven layers.² The highest layer, layer 7, is the application layer and includes protocols like HTTP. HTTP is used to transport messages containing content including HTML, XML, SOAP and web services. For the purposes of this paper, we will focus on application attacks carried by HTTP.

Traditional firewalls can be ineffective against HTTP-carried attacks. The application attacker uses valid HTTP requests over well-known ports so network firewalls will allow the attack traffic, by design, because it is *good* traffic when viewed at the network layer. What is *bad* is not the HTTP request itself but the data contained within the request. Often this harmful data is user input that is specially formatted or organized to change the behavior of your application. Application attacks can allow unrestricted access to databases, execute arbitrary system commands, or alter website content.

Proper application security prevents the user from altering the behavior of your application.

1.2.1 Common conditions that can lead to poor Application Security

- 🔥 Application security requirements, if defined, are usually seen as non-functional requirements, negative statements (you will not ...), or vague expectations.
- 🔥 Application security testing is only performed as part of an audit process.
- 🔥 Application requirement and design teams view security as a network or IT team issue.
- 🔥 Typical testing procedures are focused on proper functional behavior.
- 🔥 Only the few “security experts” in the organization are aware of application security threats.

² International Organization for Standardization (www.iso.org)

🔥 Application security testing is limited to small windows where “good guys attempt to do what potential bad guys might do.”

2. APPLICATION SECURITY THREATS

Security threats will define what security technologies can be most effectively used to defend your application. It is often best to work with generic countermeasure concepts before selecting a specific technology. Doing so will help ensure the best technologies are chosen on their merits and not because they feature the latest buzzword.

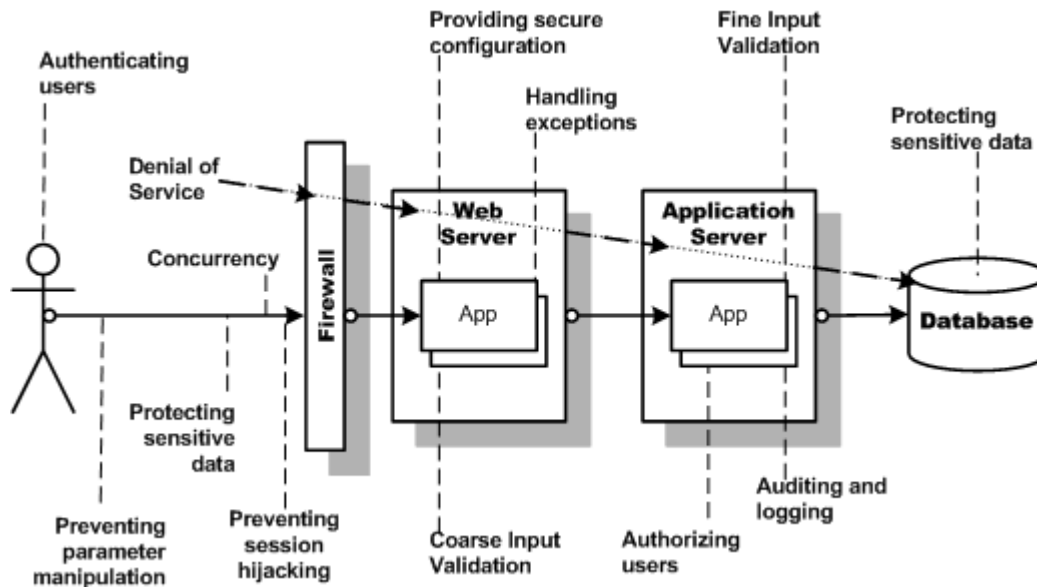


Figure 1: Common web application security concerns

A list of common threats and possible solutions is listed below. The specific threats to your application will be different.

2.1 IMPERSONATION

Anytime someone requests access to non-public information, an organization needs to make sure they are who they say they are. In general, you can prevent impersonation by using stringent authentication. You can also defend against impersonation by keeping credential information safe.

Examples	<ul style="list-style-type: none"> 🔥 An attacker typing in a different user's credentials 🔥 Changing the contents of a cookie or parameter to pretend that he/she is a different user or that the cookie comes from a different server
Common Mistakes	<ul style="list-style-type: none"> 🔥 Using communications-based authentication to allow access to any user's data 🔥 Using unencrypted credentials that an eavesdropper can capture and reuse 🔥 Storing credentials in cookies or parameters 🔥 Using self-designed or unproven authentication methods 🔥 Client software is not allowed to authenticate the host when required. 🔥 Using an authentication from the wrong trust domain

Possible Solutions	<ul style="list-style-type: none"> 🚫 Operating system supplied frameworks (e.g., Kerberos) 🚫 Encrypted tokens such as session cookies 🚫 Using digital signatures
---------------------------	---

Table 1: Impersonation threats

2.2 TAMPERING

Tampering means changing or deleting a resource without authorization.

Examples	<ul style="list-style-type: none"> 🚫 Defacing a website 🚫 Altering data in transit
Common Mistakes	<ul style="list-style-type: none"> 🚫 Trusting data sources without validation 🚫 Sanitizing input to prevent the execution of unwanted code 🚫 Running with escalated privileges 🚫 Sensitive data is left unencrypted
Possible Solutions	<ul style="list-style-type: none"> 🚫 Using operating system security to lock down files, directories and other resources 🚫 Validating your data 🚫 Hash and signed data in transit (SSL or IPsec)

Table 2: Tampering threats

2.3 REPUDIATION

Repudiation is the idea of denying that an action occurred. A repudiation attack tries to destroy, hide or alter evidence.

Examples	<ul style="list-style-type: none"> 🚫 Deleting logs 🚫 Using impersonation to request changes
Common Mistakes	<ul style="list-style-type: none"> 🚫 Poor or missing authorization and authentication 🚫 Improper logging 🚫 Allowing sensitive information on unsecured communication channels
Possible Solutions	<ul style="list-style-type: none"> 🚫 Stringent authentication, audits, transaction records, logs, or digital signatures

Table 3: Repudiation threats

2.4 INFORMATION DISCLOSURE

Information disclosure simply means revealing private information. The severity will depend upon the amount and sensitivity of the information disclosed. Data tampering is the ability to modify disclosed information.

Examples	<ul style="list-style-type: none"> 🚫 Stealing passwords 🚫 Obtaining credit card information or other similar personally identifiable information (PII) 🚫 Obtaining information about the application source and/or its host machines
Common Mistakes	<ul style="list-style-type: none"> 🚫 Allowing an authenticated user access to other users' data 🚫 Allowing sensitive information on unsecured communication channels 🚫 Poor selection of encryption algorithms and keys

Possible Solutions	<ul style="list-style-type: none"> 🔥 Storing information on a session (transitory) rather than permanent basis 🔥 Using hashing and encryption whenever possible 🔥 Matching user data to user authentication
---------------------------	--

Table 4: Information disclosure threats

2.5 DENIAL OF SERVICE (DOS)

A DoS attack causes an application to be less available than it should be. DoS attacks take two forms: 1) Flooding, where many messages are sent to overwhelm a server; 2) Lockout, where the request forces the server to take a long time to respond by consuming resources, or preventing resources from being available.

DoS attacks can take place at any level of the OSI model. They are relatively easy to mount, and difficult to defend against.

Examples	<ul style="list-style-type: none"> 🔥 Sending the application too many simultaneous requests 🔥 Sending requests that cause the application to restart or take a long time to process
Common Mistakes	<ul style="list-style-type: none"> 🔥 Pleasing too many or conflicting applications on a single server 🔥 Incomplete unit testing
Possible Solutions	<ul style="list-style-type: none"> 🔥 Filtering packets using a firewall 🔥 Using a load balancer to throttle the number of requests from a single source 🔥 Using asynchronous protocols to handle computationally intensive requests -- proper error recovery

Table 5: Denial of Service (DoS) threats

2.6 ELEVATION OF PRIVILEGE

An elevation of privilege means receiving more permissions than normally assigned.

Examples	<ul style="list-style-type: none"> 🔥 User gains administrative rights 🔥 Employee gains access to a manager role
Common Mistakes	<ul style="list-style-type: none"> 🔥 Running web server processes as "root" or "administrator" 🔥 Errors in coding allow buffer overflows, placing the application into an elevated debug state
Possible Solutions	<ul style="list-style-type: none"> 🔥 Using a least privilege context whenever possible 🔥 Using type safe languages and compiler options to prevent or control buffer overflows

Table 6: Elevation of privilege threat

3. REVIEWING SECURITY CONCERNS FOR EXISTING APPLICATIONS

Application security errors are created, discovered and fixed just like any other application error. This paper focuses on improving the security-specific processes that you use when creating applications. General SDLC process improvements are outside of the scope of this paper.

Appendix D: Using CMMI to improve Application Security,” is a logical place to start if you need to make general process improvements in addition to security.

Nearly every organization has existing legacy applications and systems that are currently deployed and need to be protected. The effort to assess and manage the infrastructure is demanding, expensive and ongoing. There are numerous books and web references on how to best manage security for an established application. In fact, much of this material assumes that fixing the application is outside of your control. While this paper’s proposed security guidelines will discuss benefits in the context of existing applications, applying these guidelines to new or re-engineered applications can derive an even greater value.

You should already be doing the following for established applications:

3.1 DISCOVERY AND BASELINES

- A complete inventory of all application and systems. This includes technical information (IP, DNS, OS used, etc...) as well as business information, such as who authorized the deployment and who needs to be notified if you need to pull the plug.
- Systems scanned for common vulnerabilities and exploits. The OS, web server and other third-party products you rely upon need to be checked for known attacks. These attacks are normally published and readily available. Ideally, prior to loading your application on a server, it was patched, hardened and scanned.
- Applications scanned for vulnerabilities to known attacks. Application assessments look at the HTTP requests your application uses and tries to manipulate the data. This assessment is usually based on reviewing the application as a black-box.
- Application authentication and user rights management tested.
- Terminate all unknown services.

3.2 RISK ASSESSMENT AND ASSIGNMENT

- Rate applications and systems for risk. Data stores, access control, user provisioning and rights management should be highlighted.
- Prioritize application vulnerabilities discovered during the assessments. Appendix E: Microsoft’s DREAD-based risk scoring is one possible framework.
- Review organizational, industry and governmental policy compliance. What is or is not acceptable must be defined.

3.3 SHIELDING AND DAMAGE CONTROL

- Patches, if available, may be applied to the application and/or infrastructure.
- Sometimes you cannot or will not be able to fix security issues in an application. In these cases, the security flaw should be shielded to prevent or minimize the exposure. An application firewall can be used to shield, or the application may be restricted, disabled or relocated.

3.4 ONGOING MONITORING AND REVIEW

Assessments are scheduled as part of the documented change management processes. This closes the circle by beginning a new discovery stage.

4. RESOLVING ERRORS THROUGHOUT THE SDLC

4.1 COST OF FIXING ERRORS IN THE SDLC

Every application faces pressures on the SDLC, such as competitive time to market demands, growing application complexity and increasing business risks. Costs escalate dramatically the longer application errors go undiscovered. Table 7 is taken from a 2002 NIST study. Following is the direct link:

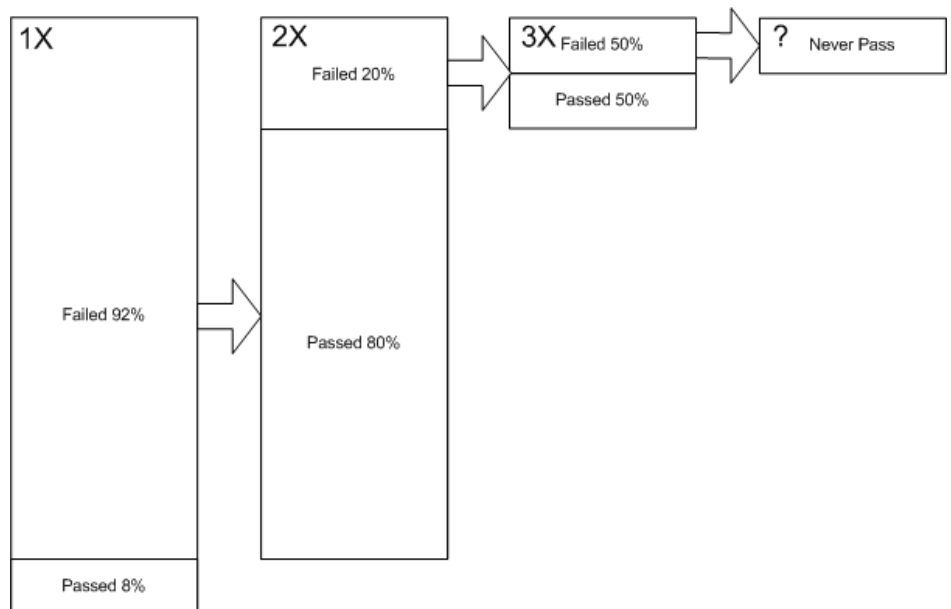
<http://www.nist.gov/director/prog-ofc/report02-3.pdf>

	Found in Design	Found in Coding	Found in Integration	Found in Beta	Found in GA
Design Errors	1x	5x	10x	15x	30x
Coding Errors		1x	10x	20x	30x
Integration Errors			1x	10x	20x

Table 7: Relative costs based on time lapse between error creation and discovery

When a design error is found in GA, the expense is 30 times what it would have cost to fix the error during the design phase. This is just the application team cost of fixing the error. The study did not factor in other costs such as lost market share, reputation or customer satisfaction.

A study conducted by Sanctum (acquired by Watchfire in 2004) of over 100 applications at large corporate and government sites places some hard numbers on security failure rates. The study found that 92 percent of all applications failed security testing conducted in the integration or production stages. The average time to fix the errors was 2.5 months, and the cost to the business team averaged \$25M. When the failed applications were tested again, 20 percent (16 percent of the total) security testing failed a second time. Half of these re-failed applications (8 percent of the total) never passed.



Given the likelihood of failed security testing and the cost of discovery errors late in the SDLC, it makes sense to improve security testing throughout the lifecycle and find security errors early.

4.2 TYPES OF ERRORS IN THE SDLC PROCESS

Before we discuss how we can find and fix errors, let's review the types of errors we are trying to correct based on where in the SDLC they are typically generated.

4.2.1 Requirements Stage Errors

The requirements team is often given general security expectations that their application is expected to adhere to. But when the requirements team is unaware of application security threats, it is unlikely that specific application requirements will be given to the design team, including requirements to scan for specific application threats.

4.2.2 Design Stage Errors

If you have a good set of requirements, where can the design team go wrong? They, too, need to be aware of application security threats. Poor or mismatched technology selection can cause the design team to erroneously believe they have fulfilled a security requirement. That lack of security knowledge can mean that a testing framework is missing, incomplete or ineffective.

4.2.3 Coding Stage Errors

You now have a good design, and the developer is expected to build to that design. What errors do they introduce? Instead of writing brand new code, they reuse flawed code, or generate code using a security unaware IDE wizard. They may not do proper data validation or they may not use the security features of the application's selected framework properly.

4.2.3 Late Stage Errors

In most organizations, application security knowledge is typically held by a few people that are part of "tiger" or "red" security audit teams. These teams will be scheduled in limited windows to look at an application late in the SDLC, with the hope that they will catch security errors before the application reaches the user.


Centralizing this function is done because a good white-hat attacker is rare (read *expensive*), and these teams often provide mandated audits of production systems. As we will see, this approach is a bottleneck, expensive and ineffective in finding the full range of application security errors.

4.3 GENERAL APPROACHES TO APPLICATION SECURITY TESTING

Application security errors are just that: errors. While it may take special knowledge to create the tests to discover security errors, once the security error, is found, the process you use to fix it is exactly the same as any other error.

4.3.1 Manual Testing

Penetration or security acceptance testing is often completed by a small set of security experts. Often these tests are done with the assistance of known tools and scripts.

Pros	 Generates well-targeted tests to specific application functions
-------------	---

Cons	<ul style="list-style-type: none"> 🚫 Limited experts perform security testing 🚫 Human error 🚫 High reoccurring expense 🚫 Time constraints limit the application coverage
-------------	--

4.3.2 Automated testing

Automated testing is typically built in one of two ways. 1) A bottom-up approach where individual functions or methods have specific tests typically built by the code developer. 2) A top-down approach where QA teams build tests from an end-user perspective.

Automated testing requires greater overhead to create and maintain than manual testing. This expense is normally offset by quality improvements, reduced effort for acceptance testing and improved iterative development processes.

4.3.3 Black-box testing

Black box, sometimes called “system testing,” is a top-down approach. The assumption is that you know nothing about how the “inside” of the application works. Your knowledge of the application is limited to seeing the application’s input and output. This is the most common form of security testing, and is used by auditors, pen testers and hackers. The test(s) consist of modifying “normal” user input in an attempt to get the application to behave in an unexpected way.

Pros	<ul style="list-style-type: none"> 🚫 Little or no application knowledge required 🚫 Well-established tools for automation of testing
Cons	<ul style="list-style-type: none"> 🚫 Testing can only be executed when all of the pieces of the application are ready for testing (typically in a late-staging or production environment). 🚫 User input mutations may result in a large number of transactions. Ignoring or reversing the results of these transactions is often problematic for production systems. 🚫 Because of limited visibility into the application, sometimes flaws are not discovered.

4.3.4 White-box

White-box, sometimes called “source testing,” tests the individual components of your application. Often this testing is preformed at the method or function level. This testing is performed to show errors in specific functions, and is often combined with code scanning tools and peer reviews.

Pros	<ul style="list-style-type: none"> 🚫 Well-defined discovery for flaws in tested functions 🚫 Established integrations with developer IDEs
Cons	<ul style="list-style-type: none"> 🚫 Because of focus at the source level, this type of testing will not discover requirement and design flaws. 🚫 Poor discovery of security errors since many attacks involve multiple components or have specific timing not covered by unit testing 🚫 Tests are often written by the same person writing the code. If the developer is not security aware, he will not know what tests he needs.

4.3.5 Gray-box testing (Using an Application Defined Framework)

A framework combines both black and white-box testing. The normal motivation to create a gray-box testing framework is to create the application state and event testing that is unavailable in commercial testing tools. Appendix F: Event-Driven Security Testing: shows one possible framework design.

Pros	<ul style="list-style-type: none"> 🔥 The most comprehensive method combining both system and unit level testing 🔥 Action driven tests can provide state and timing based tests. 🔥 Agents and/or proxies can be used for cause/effect based testing. 🔥 Framework can be built to allow production based audit testing without impacting production data. 🔥 The framework can monitor data flows through the application.
Cons	<ul style="list-style-type: none"> 🔥 Must be defined as part of requirements and design phases 🔥 The effort to build test framework is often as large as the application to be tested.

5. DEFINE YOUR APPROACH

Watchfire recommends that every organization complete tasks in four categories: 1) Security Awareness, 2) Application Risk and Liability Categorization, 3) Zero Tolerance Enforcement and 4) Security Testing Integrated into Development process. While it is possible to achieve improvements by focusing on one to two task categories, the best results are achieved by completing all categories.

5.1 SECURITY AWARENESS

This category consists of training, communication and monitoring tasks. To effectively complete these tasks, it is recommended that organizations be prepared for a consultative or collaborative approach. One-way edits, unread reports and ignored policies are counter-productive. The following is meant to be a potential guideline only. Your individual requirements may vary.

Training	<ul style="list-style-type: none"> 🔥 Provide a half day of application security training annually for all members of the application team, including developers, QA, analysts and managers -- this training should cover what current attacks are, how they are created and what the recommended remediation process is. The training should provide information on the organization’s current security posture and feedback on the organization’s security best practices. 🔥 Every developer should attend framework specific security training. The typical length of training (1-5 days) varies by framework and sometimes can be completed as self-instruction. Every serious framework has pre-built security functions that should be mastered. 🔥 As recommend by the vendor training in the proper use of any COTS security tool selected.
Communication	<ul style="list-style-type: none"> 🔥 Security best practices guidelines are drawn from all teams and all lines of business. This document should be short (<10 pages), principal driven and applicable across the organization. 🔥 Develop processes that contain a component of peer mentoring 🔥 Every application team is assigned a liaison from the Security team to help with application requirements and design issues.

Monitoring	<ul style="list-style-type: none"> 🔥 The security posture of any application in production should be known at all times. 🔥 Security errors should be tracked through normal defect tracking and reporting infrastructures to ensure all parties have the proper visibility.
-------------------	---

Table 8: Security Awareness Tasks

5.2 APPLICATION RISK AND LIABILITY CATEGORIZATION

Every organization is faced with limited resources. Priorities must be managed, and security is no exception. We have found that organizations often have a difficult time assigning absolute dollar values to application risk, and when they do succeed in this ordering, it does not change much from the softer models such as DREAD (see Appendix E). Unless you have unlimited resources, you need to do the following tasks:

- 🔥 Define risk thresholds. Include when the security team is expected to terminate application services.
- 🔥 Categorize applications by risk factors (e.g., Internal vs. External Users or by network deployment, Internet, Intranet vs. Extranet)
- 🔥 All security scans of an application should result in a risk report that is matched against the defined risk thresholds.

5.3 ZERO TOLERANCE ENFORCEMENT

This category may sound difficult. A better title may be “Ignorance of the law is no excuse.” When your organization has a well-defined security policy, you should know prior to deployment whether your application complies or not.

- 🔥 The application team should know what tests it will need to be passed at the beginning of the project.
- 🔥 The requirements and design of the application should be formally reviewed for security issues before coding begins.
- 🔥 If there is a compelling reason for the application to not follow the organization’s security policy, an exception must be granted by the CIO as part of the design approval process.
- 🔥 Clear security policies must be in place.
- 🔥 The application team’s target defect rate for all errors should be less than one error per 10K lines of code.

5.4 SECURITY TESTING INTEGRATED INTO DEVELOPMENT PROCESS

This task group is presented last for a reason. To be effective, this group relies upon you to make significant process with the other task groups. When completed, this task group has the biggest return on investment. This task group is the only one that has significant impact on the design, development and testing of your application.

- 🔥 The security tests an application needs to pass must be an explicit functional requirement.
- 🔥 The test framework must be able to be run on demand.
- 🔥 The application test framework should include unit and system tests as well as any test required to address threats to the application.

- The test framework should be automated.
- The design of the application test framework should allow for audit testing in production.
- When building the application testing framework, make sure to include testing based on an event driven model like the one defined in Appendix F: Event-Driven Security Testing.
- Although not a requirement, agile development processes like XP and SCRUM are preferred security development methodologies.
- The application design must allow for the test framework to be run during the coding, testing, integration and production stages.

6. VALIDATE YOUR METHODOLOGY

It is important to make progress on all four tracks.

6.1 SECURITY AWARENESS

The entire development team should be aware at a concept level what application security attacks are and how they operate.

Key actions:

- Annual application security training.
- Application security training can be combined with organizational specific policy training.

6.2 APPLICATION RISK AND LIABILITY CATEGORIZATION

The deliverable of this track includes a database ranking every application by risk for each application. Each team should know the status of their application, and be able to compare to other applications that are already deployed.

Key actions:

- Application risk data collected
- Application compared to corporate security baseline
- Security requirements for the application are defined based on baseline policies.

6.3 ZERO TOLERANCE ENFORCEMENT

Once the team is aware of security attacks and the application has been rated for risk, the team needs to be held accountable for a secure delivery. The key to making this work is to let the team know up front what the application is going to be tested for.

Key actions:

- Security exceptions are allowed only in the design phase and only with appropriate management approval.

- The range of security tests are defined to the application team as a requirement.

6.4 SECURITY TESTING

In this track, the results of the first three tracks are brought together. The team is aware, the risks are known and a zero tolerance policy is in place. To deliver on application quality and security in a predictable manner, automated testing tools must be deployed throughout the software development life cycle.

Key actions:

- Automated security testing tools for developers, QA teams and auditor
- Comprehensive security tests can be run at any point during the development process.

APPENDIX A: REQUIREMENT PHASE SECURITY CONSIDERATIONS

The following lists are examples of things to be considered during the application design phase. You may want to add your own considerations.

Application Environment

- 🔥 Identify, understand and accommodate the security policy of the organization
- 🔥 Recognize infrastructure restrictions (services, protocols and firewall restrictions)
- 🔥 Identify hosting environment restrictions (sub netting, VPN, sandboxing)
- 🔥 Define deployment configuration of the application
- 🔥 Define network domain structures, clustering and remote application servers
- 🔥 Identify database servers
- 🔥 Secure communication features provided by the environment are known.
- 🔥 The design addresses web farm considerations (including session state management, machine specific encryption keys, Secure Sockets Layer (SSL), certificate deployment issues and roaming profiles).
- 🔥 If SSL is used by the application, the certificate authority (CA) and the types of certificates to be used are identified.
- 🔥 The design addresses the required scalability and performance criteria.
- 🔥 Code trust level is known.

Input/Data validation

- 🔥 All input is evil.

Authentication

- 🔥 Identify all trust boundaries
- 🔥 Identity accounts and/or resources that cross trust boundaries
- 🔥 Use a policy of least-privileged accounts
- 🔥 Consider account management policies
- 🔥 When security policy mandates a strong password, the mandate is enforced.
- 🔥 Ensure that communication of user credentials is encrypted (SSL, VPN, IPsec)
- 🔥 Authentication information (tokens, cookies, tickets, etc.) is not transmitted over non-encrypted connections
- 🔥 Minimal error information is returned in the event of authentication failure

Session Management

- 🔥 Session lifetime is limited.

- ❗ Session state is protected from unauthorized access.
- ❗ Session identifiers are not passed in query strings.

APPENDIX B: DESIGN PHASE SECURITY CONSIDERATIONS

The following lists are examples of things to be considered during the application design phase. You may want to add your own considerations.

Input/Data validation

- 🔥 All input is evil.
- 🔥 Input validation is performed on a server controlled by the application.
- 🔥 Client-side input validation can be done for GUI reasons but does not supersede server side validation.
- 🔥 The design addresses potential canonicalization, SQL injection and cross-site scripting issues.
- 🔥 All entry points and trust boundaries are identified.

Authentication

- 🔥 Separate access to public and restricted areas
- 🔥 Identity accounts and/or resources that cross trust boundaries
- 🔥 Identify accounts that service or administer the application
- 🔥 Ensure that credentials accepted from users are stored securely.
- 🔥 Ensure that communication of user credentials is encrypted (SSL, VPN, IPsec)
- 🔥 The identity that is used to authenticate with the database is identified by the design.

Authorization

- 🔥 All identities that are used by the application are identified and the resources accessed by each identity are known.
- 🔥 The role design offers sufficient separation of privileges (the design considers authorization granularity).
- 🔥 The design identifies code access security requirements.
- 🔥 Privileged resources and privileged operations are identified.

Configuration Management

- 🔥 Administration interfaces are secured (strong authentication and authorization is used).
- 🔥 Remote administration channels are secured.
- 🔥 Administrator privileges are separated based on roles (for example, site content developer or system administrator).
- 🔥 Least-privileged process accounts and service accounts are used.

Sensitive Data

- 🔥 Secrets are not stored unless necessary. Alternate methods have been explored at design time.
- 🔥 Identify encryption algorithms and key sizes to store secrets securely.
- 🔥 The design identifies protection mechanisms for sensitive data that is sent over the network.

Session Management

- 🔥 SSL is used to protect authentication cookies.
- 🔥 The contents of authentication cookies are encrypted.

Cryptography

- 🔥 Encryption keys are secured.
- 🔥 Only known (good) cryptography libraries and services are used.
- 🔥 Identify the proper cryptographic algorithms and key size
- 🔥 The methodology to secure the encryption keys is identified.

Exception Management

- 🔥 Define a standard approach to structured exception handling.
- 🔥 The design identifies generic error messages that are returned to the client.

Auditing and Logging

- 🔥 Identify the level of auditing and logging necessary for the application
- 🔥 Identify the key parameters to be logged and audited
- 🔥 Identify the storage, security and analysis of the application log files
- 🔥 The design considers how to flow caller identity across multiple tiers (at the operating system or application level) for auditing.

APPENDIX C: CODING PHASE SECURITY CONSIDERATIONS

The following lists are examples of what to consider during the coding phase. You may want to add your own considerations.

Coding Practices

- 🔥 Ensure that authentication and authorization is used properly.
- 🔥 If your application demands features that force you to reduce or change default security settings, test the effects and understand the implications before making the change.
- 🔥 Do not place secrets in the code. Relying on security by obscurity does not work.
- 🔥 If you don't own it, don't trust it.
- 🔥 Don't expose information that is not needed.
- 🔥 Handle errors gracefully
- 🔥 Fail to a safe mode: do not display stack traces, or leave sensitive data unprotected.

Input/Data validation

- 🔥 All input is evil.
- 🔥 All input parameters are validated (including form fields, query strings, cookies and HTTP headers).
- 🔥 Client-side input validation can be done for GUI reasons but does not supersede server-side validation.
- 🔥 A positive validation is the model used (accept only known good input) vs. a negative model (reject known bad input).
- 🔥 Data is validated for type, length, format and range.
- 🔥 Output that contains input is properly HTML- or URL-encoded.

Authentication

- 🔥 When passwords are stored, they are stored as digests (with salt).
- 🔥 Minimal error information is returned in the event of authentication failure
- 🔥 HTTP header information is not relied on to make security decisions.

Authorization

- 🔥 The application's database login is restricted to access-specific stored procedures and can not access tables directly.
- 🔥 Access to system level resources is restricted.

Configuration Management

- 🔥 Configuration stores are secured.
- 🔥 Configuration secrets are not held in plain text in configuration files.
- 🔥 Least-privileged process accounts and service accounts are used.

Sensitive Data

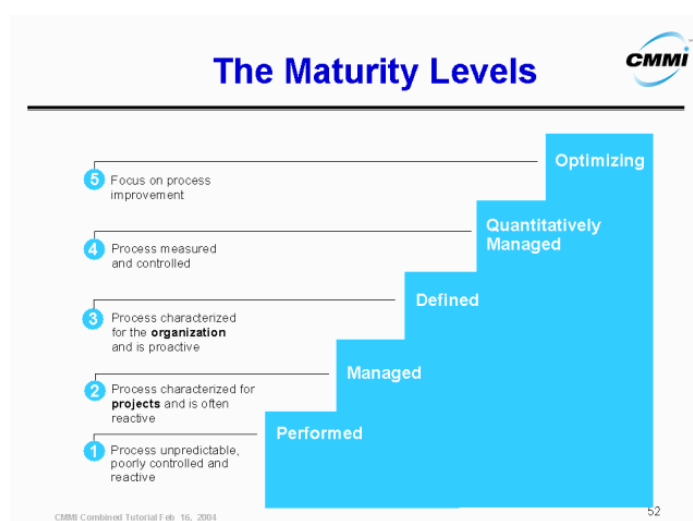
- ❖ Secrets are not stored in code.
- ❖ Secrets are not stored unless necessary. (Alternate methods have been explored at the design phase.)
- ❖ Database connections, passwords, keys or other secrets are not stored in plain text.
- ❖ Sensitive data is not logged in clear text.
- ❖ Sensitive data is not stored in cookies or transmitted as a query string or form field.

Exception Management

- ❖ Minimal information is disclosure in case of an exception.
- ❖ Sensitive data is not logged.

APPENDIX D: USING CMMI TO IMPROVE APPLICATION SECURITY

The Software Engineering Institute (SEI) at Carnegie Mellon has defined CMMI (Capability Maturity Model® Integration) <http://www.sei.cmu.edu/cmmi/>. A lot has been written about CMMI, but we are only going to discuss some basic concepts here. The website is a starting point to many detailed lectures and books on the subject, including their Team Software Process (TSP) and Personal Software Process (PSP).



The CMMI best practices enable organizations among other things, to explicitly link management and engineering activities to business objectives, and to expand the scope of and visibility into the product lifecycle and engineering activities to ensure that the product or service meets customer expectations.

It is important to keep the CMMI concepts in mind when implementing changes in your organization. Evolutionary changes are usually easier to implement than revolutionary ones. And while the Optimizing level provides the most predictability, it also contains the most overhead. The interesting claim by CMMI proponents is that the more mature

your software processes are, the earlier, and more likely, you are to find and address application errors.

Table 9: CMMI-defined application security activities shows typical application security activities for each level.

Optimizing	<ul style="list-style-type: none"> Striving for continuous improvement best practices pulled from process metrics
Quantitatively Managed	<ul style="list-style-type: none"> Coordinated application security reviews are completed for all application stages. The organization has published standards for secure application development.
Defined	<ul style="list-style-type: none"> Security reviews are performed at the individual application development stages. Security awareness is provided for all application team personnel. Predetermined application security policy is enforced for the entire organization.
Managed	<ul style="list-style-type: none"> Security audits are done on a regular basis. Audit teams and QA use automated scanning tools prior to deployment. The organization has an application security policy statement, but exceptions are permitted.
Performed	<ul style="list-style-type: none"> Audit or "tiger" teams will perform periodic infrastructure penetration tests. The organization has an application security policy statement, but enforcement is limited.

Table 9: CMMI-defined application security activities

APPENDIX E: MICROSOFT'S DREAD-BASED RISK SCORING

One of the problems with a simplistic rating system is that team members usually will not agree on ratings. To help solve this, add new dimensions that help determine what the impact of a security threat really means. At Microsoft, the DREAD model is used to help calculate risk. By using the DREAD model, you arrive at the risk rating for a given threat by asking the following questions:

- 🔥 **Damage potential:** How great is the damage if the vulnerability is exploited?
- 🔥 **Reproducibility:** How easy is it to reproduce the attack?
- 🔥 **Exploitability:** How easy is it to launch an attack?
- 🔥 **Affected users:** As a rough percentage, how many users are affected?
- 🔥 **Discoverability:** How easy is it to find the vulnerability?

When you clearly define what each value represents for your rating system, it helps avoid confusion. Table 10 shows an example of a rating table that can be used by team members when prioritizing threats. Consider using a finer (1-10) scale to apply this scoring across an entire organization.

Rating	High (3)	Medium (2)	Low(1)
Damage Potential	The attacker can subvert the system's security; get full trust authorization; run as administrator; upload content.	Disclose sensitive information	Disclose trivial information
Reproducibility	The attack can be reproduced every time and does not require a timing window.	The attack can be reproduced, but only with a timing window or a particular race situation.	The attack is very difficult to reproduce, even with knowledge of the security hole.
Exploitability	A novice programmer could make the attack in a short time.	A skilled programmer could make the attack, and then repeat the steps.	The attack requires an extremely skilled person and in-depth knowledge every time to exploit.
Affected users	All users, default configuration, key customers	Some users, non-default configuration	Very small percentage of users, obscure feature; affects anonymous users
Discoverability	Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.	The vulnerability is in a seldom-used part of the product, and only a few users should come across it. It would take some thinking to see malicious use.	The bug is obscure, and it is unlikely that users will work out damage potential.

Table 10: DREAD Rating Table

Table 11 shows a possible DREAD score of the alleged use of the “T-mobile Acct. PW Reset Exploit” in the Paris Hilton T-Mobile hack. The bolding of keywords is provided for clarity and emphasizes that even if a 1-10 finer grain scoring was used this attack would still score very high, if not perfectly. The T-Mobile response, appropriately, was to disable the affected function until the error was fixed.

Criteria	Score	Comments
Damage	3	Complete user account access and management is allowed.
Reproducibility	3	The attack works at will .
Exploitability	3	Browser-driven cut and paste operation
Affected users	3	All online T-Mobile users
Discoverability	3	Exploit receives mass media coverage and instructions are posted on well-indexed Internet websites.
	15	HIGH

Table 11: “T-mobile Acct. PW Reset Exploit” DREAD rating (5-7 low) (8-11 Medium) (12-15 High)

High, Medium, and Low Ratings

You can use a simple High, Medium or Low scale to prioritize threats. If a threat is rated as High, it poses a significant risk to your application and needs to be addressed as soon as possible. Medium threats need to be addressed, but with less urgency. You may decide to ignore Low threats depending upon how much effort and cost is required to address the threat.

Additional Microsoft Resources

Microsoft provides a number of threat model resources including a free downloadable tool called simply the “Threat Model Tool.” This tool uses a categorization method called STRIDE.

APPENDIX F: EVENT-DRIVEN SECURITY TESTING

System testing is an event driven process.

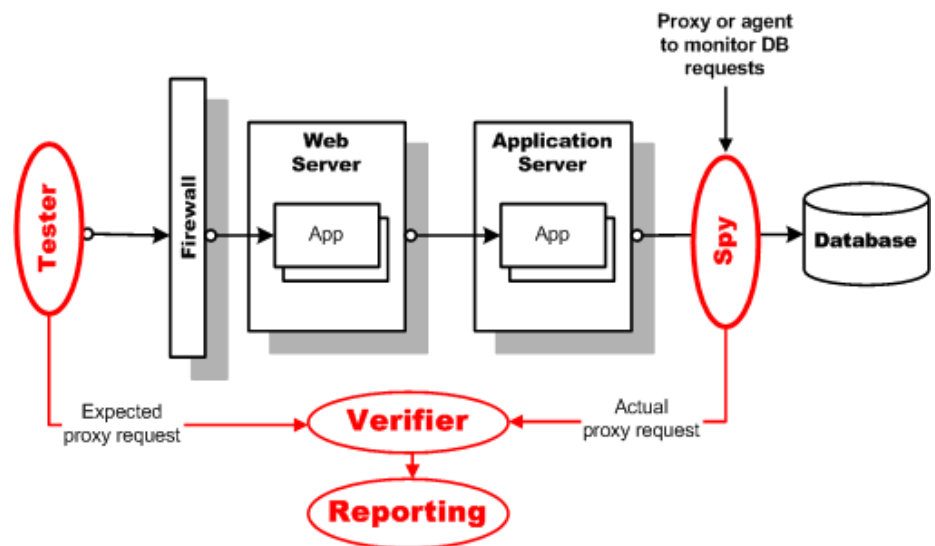
- 1) The user makes a request.
- 2) Your application responds.
- 3) The response is compared to an expected or previously stored response.
- 4) You pass or fail.

System tests are great when you have a controlled environment and know all possible user requests and responses.

So what happens when your application does not control all of the services it uses? What about when you want to test if an authenticated user is unable to access another user's data? Often you are reduced to a set of manual tests. As an example, consider the simple application shown below.

At a generic level, when you are concerned about protecting sensitive data you can write either a unit or system test to ensure that data is retrieved from the database. But how do you test if User A can retrieve User B's data? How do you test that poor user input does not impact backend processes? Manual testing is always a time consuming (and expensive) option. A better approach is to create a testing framework that is designed as part of the application.

The **Tester** generates a test and knows what the request of the **Spy** proxy or agent should be and informs the **Verifier** which compares the expected request to the actual request received by the **Spy** fronting the database.



The example shows a database as the backend component, but in reality, any service, email, XML or legacy service can be fronted.

The implementation of the code to review requests is application dependant. Possible design approaches for the **Spy** component could be: a mock data access object (DAO), proxy, sniffer, or as a class that inherits from the fronted service, are all possible solutions. The concept is that you create code specifically for testing that is inserted into the data stream. The inserted code is aware of the testing you want to perform and will report data as needed by the testing framework.

Coordinating the testing objects shown in red allows for complete fine grain control of a full range of tests and is available using black or white-box testing alone.

7. FURTHER READING

1. Non-profit Agile Alliance development methodology
<http://www.agilealliance.com/home>
2. Discussion on SCRUM agile development
<http://www.controlchaos.com/>
3. XP agile development
<http://c2.com/cgi/wiki> and <http://www.extremeprogramming.org/>
4. DHS report on improving security in the application lifecycle
<http://www.cyberpartnership.org/SDLCFULL.pdf>
5. Software Engineering Institute (SEI) at Carnegie Mellon
<http://www.sei.cmu.edu/cmmi/>
6. Testing based on JUnit
<http://junit.org/>
7. Example testing frameworks
<http://wiley.com/compbooks/javatesting/>
8. Extreme Programming Pocket Guide
<http://www.oreilly.com/catalog/extprogpg>
9. Grady Booch's software architecture handbook
<http://www.booch.com/architecture/blog.jsp>
10. The Open Web Application Security Project
<http://www.owasp.org>
11. OASIS WAS-XML TC
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=was

ABOUT WATCHFIRE

Watchfire provides software and services to manage online risk. More than 250 enterprise organizations and government agencies, including AXA Financial, SunTrust, Nationwide Building Society, Boots PLC, Veterans Affairs and Dell, rely on Watchfire to monitor, manage, improve and secure all aspects of the online business including security, privacy, quality, accessibility, corporate standards and regulatory compliance. Watchfire's alliance and technology partners include IBM Global Services, PricewaterhouseCoopers, TRUSTe, Microsoft, Interwoven, EMC Documentum and Mercury Interactive. Watchfire is headquartered in Waltham, MA. For more information, please visit www.watchfire.com.