OWASP
The Open Web Application Security Project
http://www.owasp.org

The Ten Most Critical Web Application Security Vulnerabilities

2004 Update

January 27th, 2004

Commentary

"With new vulnerabilities announced almost weekly, many businesses may feel overwhelmed trying to keep current. But there is help in the form of consensus lists of vulnerabilities and defenses.

"The Open Web Application Security Project has produced a similar list of the 10 most critical Web application and databases security vulnerabilities and the most effective ways to address them. Application vulnerabilities are often neglected, but they are as important to deal with as network issues. If every company eliminated these common vulnerabilities, their work wouldn't be done, but they, and the Internet, would be significantly safer."

- ***J. Howard Beales, III, Director of the Federal Trade Commission's Bureau of Consumer Protection, before the Information Technology Association of America's Internet Policy Committee, Friday, December 12, 2003***

"Misconfiguration, inattention, and flawed software can spell disaster on the Internet. One of the primary areas of vulnerability is through WWW connections. By design, WWW services are intended to be open and accepting, and usually act as an interface to valuable resources. As such, it is critical that these services be secured. But with hundreds of potential vulnerabilities it can be overly daunting to decide where to start applying defensive measures. The OWASP Top 10 provides a consensus view of the most significant and likely vulnerabilities in custom WWW applications. Organizations should use it to focus their efforts, giving them confidence they are addressing those areas that will have the most impact on securing their applications."

- ***Eugene H. Spafford, Professor of Computer Sciences, Purdue University and Executive Director of the Purdue University Center for Education and Research in Information Assurance and Security (CERIAS)***

"This 'Ten-Most-Wanting' List acutely scratches at the tip of an enormous iceberg. The underlying reality is shameful: most system and Web application software is written oblivious to security principles, software engineering, operational implications, and indeed common sense."

- ***Dr. Peter G. Neumann, Principal Scientist, SRI International Computer Science Lab, Moderator of the ACM Risks Forum, Author of "Computer-Related Risks"***

"This list is an important development for consumers and vendors alike. It will educate vendors to avoid the same mistakes that have been repeated countless times in other web applications. But it also gives consumers a way of asking vendors to follow a minimum set of expectations for web application security and, just as importantly, to identify which vendors are not living up to those expectations"

- ***Steven M. Christey, Principal Information Security Engineer and CVE Editor, Mitre***

"The OWASP Top Ten shines a spotlight directly on one of the most serious and often overlooked risks facing government and commercial organizations. The root cause of these risks is not flawed software, but software development processes that pay little or no attention to security. The most effective first step towards creating a security-aware culture in your organization is immediately adopting the Top Ten as a minimum standard for web application security."

- ***Jeffrey R. Williams, Aspect Security CEO and OWASP Top Ten Project Leader***

"There is no silver bullet for web security despite what some technology companies would have you believe. Solving this complex and challenging problem is about having great people, great knowledge, great education, great business process and using great technology. The OWASP Top Ten creates a palatable well ordered and well thought out list from where you can start to understand your security posture (or that of your service providers) and plan the use of your valuable resources accordingly."

- ***Mark Curphey, Founder of OWASP and Director of Application Security Consulting, Foundstone Strategic Security.***

"From web pages to back office number crunching, almost all organizations acquire applications code, many people write it, and everybody uses it. But flaws continue to be found in applications, even after nearly fifty years of programming experience. Worse, the same kinds of flaws appear over and over again. This failure to learn from not only our mistakes but also those of our parents' generation creates far too many vulnerabilities for potential attack. It is no wonder that attacks against applications are on the rise.

In compiling this list of the ten most critical applications code flaws, the OWASP has performed a real service to developers and users alike by focusing attention on common weaknesses and what can be done about them. Now it is up to software development organizations, programmers, and users to apply the thoughtful guidance presented here."

■ ***Dr. Charles P. Pfleeger, CISSP, Master Security Architect, Cable & Wireless, Author of "Security in Computing"***

"The new ROI is security. Companies must be able to provide trusted web applications and web services to their trading partners; who are requiring both secure technology and traditional indicia of financial security, such as insurance. It doesn't stop there though; bad security places a companies' data and applications at risk and hence its viability as a commercial entity. Disappointing your customers is one thing, trying to come back from the loss or corruption of your own systems is quite another."

■ ***Robert A. Parisi, Jr., Senior VP and Chief Underwriting Officer, AIG eBusiness Risk Solutions***

"Web developers need to know that the degree to which business applications and customer data are protected from the hostile Internet is directly determined by how securely they've written their code. The OWASP Top Ten list is a great way to understand how to code defensively and avoid the security pitfalls that plague Web applications."

■ ***Chris Wysopal, Director of Research & Development, @stake, Inc.***

"The healthcare industry has a critical need to provide secure web applications that protect users' privacy. The OWASP Top Ten will help healthcare organizations evaluate the security of web application products and solutions. Any healthcare organizations using applications that contain these flaws may have difficulty complying with the HIPAA regulations."

■ ***Lisa Gallagher, Senior VP, Information and Technology Accreditations, URAC***

"As more and more companies turn to offshore outsourcing for Web application development, secure coding enforcement must jump to the top of the executive priority list. The OWASP Top Ten Project accurately articulates the top application vulnerabilities, providing C-level executives a goldmine of information useful in setting security policy guidelines for Web applications."

■ ***Stuart McClure, President and CTO of Foundstone Inc.***

"Application security considerations are often treated as the domain of specialists, to be applied after coding is done and all other business requirements are satisfied. At Oracle we have found that the task of securing applications is most likely to succeed if security is built in from the start. Developers must have a basic understanding of security vulnerabilities and how to avoid them, and release managers must explicitly check that security requirements have been satisfied before product can ship. The OWASP Top Ten List is an excellent starting point for companies to build security awareness among their developers, and to prepare security criteria for application deployment."

■ ***John Heimann, Director of Security at Oracle Corp.***

"Many people have seen top ten lists detailing the "biggest network security risks". Those lists only point out flaws in third party software that you may or may not be running on your network. Applying the information in this list will keep the software you develop off those other lists!"

■ ***John Viega, Chief Scientist, Secure Software Inc., co-author of "Building Secure Software"***

Table of Contents

Introduction

The Open Web Application Security Project (OWASP) is dedicated to helping organizations understand and improve the security of their web applications and web services. This list was created to focus corporations and government agencies on the most serious of these vulnerabilities. Web application security has become a hot topic as companies race to make content and services accessible though the web. At the same time, hackers are turning their attention to the common weaknesses created by application developers.

When an organization puts up a web application, they invite the world to send them HTTP requests. Attacks buried in these requests sail past firewalls, filters, platform hardening, and intrusion detection systems without notice because they are inside legal HTTP requests. Even "secure" websites that use SSL just accept the requests that arrive through the encrypted tunnel without scrutiny. **This means that your web application code is part of your security perimeter**. As the number, size and complexity of your web applications increases, so does your perimeter exposure.

The security issues raised here are not new. In fact, some have been well understood for decades. Yet for a variety of reasons, major software development projects are still making these mistakes and jeopardizing not only their customers' security, but also the security of the entire Internet. There is no "silver bullet" to cure these problems. Today's assessment and protection technology is improving, but can currently only deal with a limited sub-set of the issues at best. To address the issues described in this document, organizations will need to change their development culture, train developers, update their software development processes, and use technology where appropriate.

The OWASP Top Ten is a list of vulnerabilities that require immediate remediation. Existing code should be checked for these vulnerabilities immediately, as these flaws are being actively targeted by attackers. Development projects should address these vulnerabilities in their requirements documents and design, build, and test their applications to ensure that they have not been introduced. Project managers should include time and budget for application security activities including developer training, application security policy development, security mechanism design and development, penetration testing, and security code review.

We encourage organizations to join the growing list of companies that have **adopted the OWASP Top Ten as a minimum standard** and are committed to producing web applications that do not contain these vulnerabilities.

We have chosen to present this list in a format similar to the highly successful SANS/FBI Top Twenty List in order to facilitate its use and understanding. The SANS list is focused on flaws in particular widely used network and infrastructure products. Because each website is unique, the OWASP Top Ten is organized around particular types or categories of vulnerabilities that frequently occur in web applications. These categories are being standardized in the OASIS Web Application Security (WAS) XML Project.

This list represents the combined wisdom of OWASP experts, whose experience includes many years of application security work for governments, financial services, pharmaceuticals and manufacturing, as well as developing tools and technology. This document is designed to introduce the most serious web application vulnerabilities. There are many books and guidelines that describe these vulnerabilities in more detail and provide detailed guidance on how to eliminate them. One such guideline is the OWASP Guide available at http://www.owasp.org.

The OWASP Top Ten is a living document. It includes instructions and pointers to additional information useful for correcting these types of security flaws. We update the list and the instructions as more critical threats and more current or convenient methods are identified, and we welcome your input along the way. This is a community consensus document – your experience in fighting attackers and in eliminating these vulnerabilities can help others who come after you. Please send suggestions via e-mail to topten@owasp.org with the subject "OWASP Top Ten Comments."

OWASP gratefully acknowledges the special contribution from **Aspect Security** for their role in the research and preparation of this document.

aspect security

http://www.aspectsecurity.com

Background

The challenge of identifying the "top" web application vulnerabilities is a virtually impossible task. There is not even widespread agreement on exactly what is included in the term "web application security." Some have argued that we should focus only on security issues that affect developers writing custom web application code. Others have argued for a more expansive definition that covers the entire application layer, including libraries, server configuration, and application layer protocols. In the hopes of addressing the most serious risks facing organizations, we have decided to give a relatively broad interpretation to web application security, while still keeping clear of network and infrastructure security issues.

Another challenge to this effort is that each specific vulnerability is unique to a particular organization's website. There would be little point in calling out specific vulnerabilities in the web applications of individual organizations, especially since they are hopefully fixed soon after a large audience knows of their existence. Therefore, we have chosen to focus on the top classes, types, or categories of web application vulnerabilities.

In the first version of this document, we decided to classify a wide range of web application problems into meaningful categories. We studied a variety of vulnerability classification schemes and came up with a set of categories. Factors that characterize a good vulnerability category include whether the flaws are closely related, can be addressed with similar countermeasures, and frequently occur in typical web application architectures. In this version we are introducing a refined scheme. This has been developed with our continued work on the OASIS WAS technical committee in which we will be describing a Thesaurus of issues from which security researchers can describe signatures in an XML format.

To choose the top ten from a large list of candidates has its own set of difficulties. There are simply no reliable sources of statistics about web application security problems. In the future, we would like to gather statistics about the frequency of certain flaws in web application code and use those metrics to help prioritize the top ten. However, for a number of reasons, this sort of measurement is not likely to occur in the near future.

We recognize that there is no "right" answer for which vulnerability categories should be in the top ten. Each organization will have to think about the risk to their organization based on the likelihood of having one of these flaws and the specific consequences to their enterprise. In the meantime, we put this list forward as a set of problems that represent a significant amount of risk to a broad array of organizations. The top ten themselves are not in any particular order, as it would be almost impossible to determine which of them represents the most aggregate risk.

The OWASP Top Ten project is an ongoing effort to make information about key web application security flaws available to a wide audience. We expect to update this document anually based on discussion on the OWASP mailing lists and feedback to topten@owasp.org.

What's New?

OWASP has come a long way since the last top was released in January 2003. This update incorporates all the discussion and up to date views, opinions and debate in the OWASP community over the past 12 months. Overall, there have been minor improvements to all parts of the Top Ten, and only a few major changes:

- **WAS-XML Alignment** – One of the new projects initiated in 2003 is the Web Application Security Technical Committee (WAS TC) at OASIS. The purpose of the WAS TC is to produce a classification scheme for web security vulnerabilities, a model to provide guidance for initial threat, impact and therefore risk ratings, and an XML schema to describe web security conditions that can be used by both assessment and protection tools. The OWASP Top Ten project has used the WAS TC as a reference for reprofiling the Top Ten to provide a standardized approach to the classification of web application security vulnerabilities. The WAS Thesaurus defines a standard language for discussing web application security, and we adopt that vocabulary here.
- **Addition of Denial of Service** – The only top level category that changed was the addition of the A9 Denial of Service category to the list. Our research has shown that a broad array of organizations are susceptible to this type of attack. Based on the likelihood of a denial of service attack and the consequences if the attack succeeds, we have determined that it warrants inclusion in the Top Ten. To accommodate this new entry, we have combined last year's A9 Remote Administration Flaws into the A2 Broken Access Control category as it is a special case of that category. We believe this is appropriate, as the types of flaws in A2 are typically the same as those in A9 and require the same types of remediation.

The table below highlights the relationship between the new Top Ten, last year's Top Ten, and the WAS TC Thesaurus.

New Top Ten 2004	Top Ten 2003	New WAS Thesaurus
A1 Unvalidated Input	A1 Unvalidated Parameters	Input Validation
A2 Broken Access Control	A2 Broken Access Control (A9 Remote Administration Flaws)	Access Control
A3 Broken Authentication and Session Management	A3 Broken Account and Session Management	Authentication and Session Management
A4 Cross Site Scripting (XSS) Flaws	A4 Cross Site Scripting (XSS) Flaws	Input Validation->Cross site scripting
A5 Buffer Overflows	A5 Buffer Overflows	Buffer Overflows
A6 Injection Flaws	A6 Command Injection Flaws	Input Validation->Injection
A7 Improper Error Handling	A7 Error Handling Problems	Error Handling
A8 Insecure Storage	A8 Insecure Use of Cryptography	Data Protection
A9 Denial of Service	N/A	Availability
A10 Insecure Configuration Management	A10 Web and Application Server Misconfiguration	Application Configuration Management Infrastructure Configuration Management

The Top Ten List

The following is a short summary of the most significant web application security vulnerabilities. Each of these is described in more detail in the following sections.

Top Vulnerabilities in Web Applications		
A1	**Unvalidated Input**	Information from web requests is not validated before being used by a web application. Attackers can use these flaws to attack backend components through a web application.
A2	**Broken Access Control**	Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access other users' accounts, view sensitive files, or use unauthorized functions.
A3	**Broken Authentication and Session Management**	Account credentials and session tokens are not properly protected. Attackers that can compromise passwords, keys, session cookies, or other tokens can defeat authentication restrictions and assume other users' identities.
A4	**Cross Site Scripting (XSS) Flaws**	The web application can be used as a mechanism to transport an attack to an end user's browser. A successful attack can disclose the end user's session token, attack the local machine, or spoof content to fool the user.
A5	**Buffer Overflows**	Web application components in some languages that do not properly validate input can be crashed and, in some cases, used to take control of a process. These components can include CGI, libraries, drivers, and web application server components.
A6	**Injection Flaws**	Web applications pass parameters when they access external systems or the local operating system. If an attacker can embed malicious commands in these parameters, the external system may execute those commands on behalf of the web application.
A7	**Improper Error Handling**	Error conditions that occur during normal operation are not handled properly. If an attacker can cause errors to occur that the web application does not handle, they can gain detailed system information, deny service, cause security mechanisms to fail, or crash the server.
A8	**Insecure Storage**	Web applications frequently use cryptographic functions to protect information and credentials. These functions and the code to integrate them have proven difficult to code properly, frequently resulting in weak protection.
A9	**Denial of Service**	Attackers can consume web application resources to a point where other legitimate users can no longer access or use the application. Attackers can also lock users out of their accounts or even cause the entire application to fail.
A10	**Insecure Configuration Management**	Having a strong server configuration standard is critical to a secure web application. These servers have many configuration options that affect security and are not secure out of the box.

A1 Unvalidated Input

A1.1 Description

Web applications use input from HTTP requests (and occasionally files) to determine how to respond. Attackers can tamper with any part of an HTTP request, including the url, querystring, headers, cookies, form fields, and hidden fields, to try to bypass the site's security mechanisms. Common names for common input tampering attacks include: forced browsing, command insertion, cross site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, and hidden field manipulation. Each of these attack types is described in more detail later in this paper.

- A4 – Cross Site Scripting Flaws discusses input that contains scripts to be executed on other user's browsers
- A5 – Buffer Overflows discusses input that has been designed to overwrite program execution space
- A6 – Injection Flaws discusses input that is modified to contain executable commands

Some sites attempt to protect themselves by filtering out malicious input. The problem is that there are so many different ways of encoding information. These encoding formats are not like encryption, since they are trivial to decode. Still, developers often forget to decode all parameters to their simplest form before using them. Parameters must be converted to the simplest form before they are validated, otherwise, malicious input can be masked and it can slip past filters. The process of simplifying these encodings is called "canonicalization." Since almost all HTTP input can be represented in multiple formats, this technique can be used to obfuscate any attack targeting the vulnerabilities described in this document. This makes filtering very difficult.

A surprising number of web applications use only client-side mechanisms to validate input. Client side validation mechanisms are easily bypassed, leaving the web application without any protection against malicious parameters. Attackers can generate their own HTTP requests using tools as simple as telnet. They do not have to pay attention to anything that the developer intended to happen on the client side. Note that client side validation is a fine idea for performance and usability, but it has no security benefit whatsoever. Server side checks are required to defend against parameter manipulation attacks. Once these are in place, client side checking can also be included to enhance the user experience for legitimate users and/or reduce the amount of invalid traffic to the server.

These attacks are becoming increasingly likely as the number of tools that support parameter "fuzzing", corruption, and brute forcing grows. The impact of using unvalidated input should not be underestimated. A huge number of attacks would become difficult or impossible if developers would simply validate input before using it. Unless a web application has a strong, centralized mechanism for validating all input from HTTP requests (and any other sources), vulnerabilities based on malicious input are very likely to exist.

A1.2 Environments Affected

All web servers, application servers, and web application environments are susceptible to parameter tampering.

A1.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: Data Validation http://www.owasp.org/documentation/guide/
- modsecurity project (Apache module for HTTP validation) http://www.modsecurity.org
- How to Build an HTTP Request Validation Engine (J2EE validation with Stinger) http://www.owasp.org/columns/jeffwilliams/jeffwilliams2
- Have Your Cake and Eat it Too (.NET validation) http://www.owasp.org/columns/jpoteet/jpoteet2

A1.4 How to Determine If You Are Vulnerable

Any part of an HTTP request that is used by a web application without being carefully validated is known as a "tainted" parameter. The simplest way to find tainted parameter use is to have a detailed code review, searching for all the calls where information is extracted from an HTTP request. For example, in a J2EE application, these are the methods in the HttpServletRequest class. Then you can follow the code to see where that variable gets used. If the variable is not checked before it is used, there is very likely a problem. In Perl, you should consider using the "taint" (-T) option.

It is also possible to find tainted parameter use by using tools like OWASP's WebScarab. By submitting unexpected values in HTTP requests and viewing the web application's responses, you can identify places where tainted parameters are used.

A1.5 How to Protect Yourself

The best way to prevent parameter tampering is to ensure that all parameters are validated before they are used. A centralized component or library is likely to be the most effective, as the code performing the checking should all be in one place. Each parameter should be checked against a strict format that specifies exactly what input will be allowed. "Negative" approaches that involve filtering out certain bad input or approaches that rely on signatures are not likely to be effective and may be difficult to maintain.

Parameters should be validated against a "positive" specification that defines:

- Data type (string, integer, real, etc…)
- Allowed character set
- Minimum and maximum length
- Whether null is allowed
- Whether the parameter is required or not
- Whether duplicates are allowed
- Numeric range
- Specific legal values (enumeration)
- Specific patterns (regular expressions)

A new class of security devices known as web application firewalls can provide some parameter validation services. However, in order for them to be effective, the device must be configured with a strict definition of what is valid for each parameter for your site. This includes properly protecting all types of input from the HTTP request, including URLs, forms, cookies, querystrings, hidden fields, and parameters.

The OWASP Filters project is producing reusable components in several languages to help prevent many forms of parameter tampering. The Stinger HTTP request validation engine (stinger.sourceforge.net) was also developed by OWASP for J2EE environments.

A2 Broken Access Control

A2.1 Description

Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others. These checks are performed after authentication, and govern what 'authorized' users are allowed to do. Access control sounds like a simple problem but is insidiously difficult to implement correctly. A web application's access control model is closely tied to the content and functions that the site provides. In addition, the users may fall into a number of groups or roles with different abilities or privileges.

Developers frequently underestimate the difficulty of implementing a reliable access control mechanism. Many of these schemes were not deliberately designed, but have simply evolved along with the web site. In these cases, access control rules are inserted in various locations all over the code. As the site nears deployment, the ad hoc collection of rules becomes so unwieldy that it is almost impossible to understand.

Many of these flawed access control schemes are not difficult to discover and exploit. Frequently, all that is required is to craft a request for functions or content that should not be granted. Once a flaw is discovered, the consequences of a flawed access control scheme can be devastating. In addition to viewing unauthorized content, an attacker might be able to change or delete content, perform unauthorized functions, or even take over site administration.

One specific type of access control problem is administrative interfaces that allow site administrators to manage a site over the Internet. Such features are frequently used to allow site administrators to efficiently manage users, data, and content on their site. In many instances, sites support a variety of administrative roles to allow finer granularity of site administration. Due to their power, these interfaces are frequently prime targets for attack by both outsiders and insiders.

A2.2 Environments Affected

All known web servers, application servers, and web application environments are susceptible to at least some of these issues. Even if a site is completely static, if it is not configured properly, hackers could gain access to sensitive files and deface the site, or perform other mischief.

A2.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: Access Control: http://www.owasp.org/guide/
- Access Control (aka Authorization) in Your J2EE Application http://www.owasp.org/columns/jeffwilliams/jeffwilliams3
- http://www.infosecuritymag.com/2002/jun/insecurity.shtml

A2.4 How to Determine If You Are Vulnerable

Virtually all sites have some access control requirements. Therefore, an access control policy should be clearly documented. Also, the design documentation should capture an approach for enforcing this policy. If this documentation does not exist, then a site is likely to be vulnerable.

The code that implements the access control policy should be checked. Such code should be well structured, modular, and most likely centralized. A detailed code review should be performed to validate the correctness of the access control implementation. In addition, penetration testing can be quite useful in determining if there are problems in the access control scheme.

Find out how your website is administrated. You want to discover how changes are made to webpages, where they are tested, and how they are transported to the production server. If administrators can make changes remotely, you want to know how those communications channels are protected. Carefully review each interface to make sure that only authorized administrators are allowed access. Also, if there are different types or groupings of data that can be accessed through the interface, make sure that only authorized data can be accessed as well. If such interfaces employ external commands, review the use of such commands to make sure they are not subject to any of the command injection flaws described in this paper.

A2.5 How to Protect Yourself

The most important step is to think through an application's access control requirements and capture it in a web application security policy. We strongly recommend the use of an access control matrix to define the access control rules. Without documenting the security policy, there is no definition of what it means to be secure for that site. The policy should document what types of users can access the system, and what functions and content each of these types of users should be allowed to access. The access control mechanism should be extensively tested to be sure that there is no way to bypass it. This testing requires a variety of accounts and extensive attempts to access unauthorized content or functions.

Some specific access control issues include:

- Insecure Id's – Most web sites use some form of id, key, or index as a way to reference users, roles, content, objects, or functions. If an attacker can guess these id's, and the supplied values are not validated to ensure the are authorized for the current user, the attacker can exercise the access control scheme freely to see what they can access. Web applications should not rely on the secrecy of any id's for protection.
- Forced Browsing Past Access Control Checks – many sites require users to pass certain checks before being granted access to certain URLs that are typically 'deeper' down in the site. These checks must not be bypassable by a user that simply skips over the page with the security check.
- Path Traversal – This attack involves providing relative path information (e.g., "../../target_dir/target_file") as part of a request for information. Such attacks try to access files that are normally not directly accessible by anyone, or would otherwise be denied if requested directly. Such attacks can be submitted in URLs as well as any other input that ultimately accesses a file (i.e., system calls and shell commands).
- File Permissions – Many web and application servers rely on access control lists provided by the file system of the underlying platform. Even if almost all data is stored on backend servers, there are always files stored locally on the web and application server that should not be publicly accessible, particularly configuration files, default files, and scripts that are installed on most web and application servers. Only files that are specifically intended to be presented to web users should be marked as readable using the OS's permissions mechanism, most directories should not be readable, and very few files, if any, should be marked executable.
- Client Side Caching – Many users access web applications from shared computers located in libraries, schools, airports, and other public access points. Browsers frequently cache web pages that can be accessed by attackers to gain access to otherwise inaccessible parts of sites. Developers should use multiple mechanisms, including HTTP headers and meta tags, to be sure that pages containing sensitive information are not cached by user's browsers.

There are some application layer security components that can assist in the proper enforcement of some aspects of your access control scheme. Again, as for parameter validation, to be effective, the component must be configured with a strict definition of what access requests are valid for your site. When using such a component, you must be careful to understand exactly what access control assistance the component can provide for you given your site's security policy, and what part of your access control policy that the component cannot deal with, and therefore must be properly dealt with in your own custom code.

For administrative functions, the primary recommendation is to never allow administrator access through the front door of your site if at all possible. Given the power of these interfaces, most organizations should not accept the risk of making these interfaces available to outside attack. If remote administrator access is absolutely required, this can be accomplished without opening the front door of the site. The use of VPN technology could be used to provide an outside administrator access to the internal company (or site) network from which an administrator can then access the site through a protected backend connection.

A3 Broken Authentication and Session Management

A3.1 Description

Authentication and session management includes all aspects of handling user authentication and managing active sessions. Authentication is a critical aspect of this process, but even solid authentication mechanisms can be undermined by flawed credential management functions, including password change, forgot my password, remember my password, account update, and other related functions. Because "walk by" attacks are likely for many web applications, all account management functions should require reauthentication even if the user has a valid session id.

User authentication on the web typically involves the use of a userid and password. Stronger methods of authentication are commercially available such as software and hardware based cryptographic tokens or biometrics, but such mechanisms are cost prohibitive for most web applications. A wide array of account and session management flaws can result in the compromise of user or system administration accounts. Development teams frequently underestimate the complexity of designing an authentication and session management scheme that adequately protects credentials in all aspects of the site.

Web applications must establish sessions to keep track of the stream of requests from each user. HTTP does not provide this capability, so web applications must create it themselves. Frequently, the web application environment provides a session capability, but many developers prefer to create their own session tokens. In either case, if the session tokens are not properly protected, an attacker can hijack an active session and assume the identity of a user. Creating a scheme to create strong session tokens and protect them throughout their lifecycle has proven elusive for many developers.

Unless all authentication credentials and session identifiers are protected with SSL at all times and protected against disclosure from other flaws, such as cross site scripting, an attacker can hijack a user's session and assume their identity.

A3.2 Environments Affected

All known web servers, application servers, and web application environments are susceptible to broken authentication and session management issues.

A3.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 6: Authentication and Chapter 7: Session Management: http://www.owasp.org/guide/
- White paper on the Session Fixation Vulnerability in Web-based Applications: http://www.acros.si/papers/session_fixation.pdf
- White paper on Password Recovery for Web-based Applications -- http://fishbowl.pastiche.org/archives/docs/PasswordRecovery.pdf

A3.4 How to Determine If You Are Vulnerable

Both code review and penetration testing can be used to diagnose authentication and session management problems. Carefully review each aspect of your authentication mechanisms to ensure that user's credentials are protected at all times, while they are at rest (e.g., on disk), and while they are in transit (e.g., during login). Review every available mechanism for changing a user's credentials to ensure that only an authorized user can change them. Review your session management mechanism to ensure that session identifiers are always protected and are used in such a way as to minimize the likelihood of accidental or hostile exposure.

A3.5 How to Protect Yourself

Careful and proper use of custom or off the shelf authentication and session management mechanisms should significantly reduce the likelihood of a problem in this area. Defining and documenting your site's policy with respect to securely managing users credentials is a good first step. Ensuring that your implementation consistently enforces this policy is key to having a secure and robust authentication and session management mechanism. Some critical areas include:

- Password Strength - passwords should have restrictions that require a minimum size and complexity for the password. Complexity typically requires the use of minimum combinations of alphabetic, numeric, and/or non-alphanumeric characters in a user's password (e.g., at least one of each). Users should be required to change their password periodically. Users should be prevented from reusing previous passwords.
- Password Use - Users should be restricted to a defined number of login attempts per unit of time and repeated failed login attempts should be logged. Passwords provided during failed login attempts should not be recorded, as this may expose a user's password to whoever can gain access to this log. The system should not indicate whether it was the username or password that was wrong if a login attempt fails. Users should be informed of the date/time of their last successful login and the number of failed access attempts to their account since that time.
- Password Change Controls: A single password change mechanism should be used wherever users are allowed to change a password, regardless of the situation. Users should always be required to provide both their old and new password when changing their password (like all account information). If forgotten passwords are emailed to users, the system should require the user to reauthenticate whenever the user is changing their e-mail address, otherwise an attacker who temporarily has access to their session (e.g., by walking up to their computer while they are logged in) can simply change their e-mail address and request a 'forgotten' password be mailed to them.
- Password Storage - All passwords must be stored in either hashed or encrypted form to protect them from exposure, regardless of where they are stored. Hashed form is preferred since it is not reversible. Encryption should be used when the plaintext password is needed, such as when using the password to login to another system. Passwords should never be hardcoded in any source code. Decryption keys must be strongly protected to ensure that they cannot be grabbed and used to decrypt the password file.
- Protecting Credentials in Transit - The only effective technique is to encrypt the entire login transaction using something like SSL. Simple transformations of the password such as hashing it on the client prior to transmission provide little protection as the hashed version can simply be intercepted and retransmitted even though the actual plaintext password might not be known.
- Session ID Protection – Ideally, a user's entire session should be protected via SSL. If this is done, then the session ID (e.g., session cookie) cannot be grabbed off the network, which is the biggest risk of exposure for a session ID. If SSL is not viable for performance or other reasons then session IDs themselves must be protected in other ways. First, they should never be included in the URL as they can be cached by the browser, sent in the referrer header, or accidentally forwarded to a 'friend'. Session IDs should be long, complicated, random numbers that cannot be easily guessed. Session IDs can also be changed frequently during a session to reduce how long a session ID is valid. Session IDs must be changed when switching to SSL, authenticating, or other major transitions. Session IDs chosen by a user should never be accepted.
- Account Lists - Systems should be designed to avoid allowing users to gain access to a list of the account names on the site. If lists of users must be presented, it is recommended that some form of pseudonym (screen name) that maps to the actual account be listed instead. That way, the pseudonym can't be used during a login attempt or some other hack that goes after a user's account.
- Browser Caching – Authentication and session data should never be submitted as part of a GET, POST should always be used instead. Authentication pages should be marked with all varieties of the no cache tag to prevent someone from using the back button in a user's browser to backup to the login page and resubmit the previously typed in credentials. Many browsers now support the autocomplete=false flag to prevent storing of credentials in autocomplete caches.
- Trust Relationships – Your site architecture should avoid implicit trust between components whenever possible. Each component should authenticate itself to any other component it is interacting with unless there is a strong reason not to (such as performance or lack of a usable mechanism). If trust relationships are required, strong procedural and architecture mechanisms should be in place to ensure that such trust cannot be abused as the site architecture evolves over time.

A4 Cross-Site Scripting (XSS) Flaws

A4.1 Description

Cross-site scripting (sometimes referred to as XSS) vulnerabilities occur when an attacker uses a web application to send malicious code, generally in the form of a script, to a different end user. These flaws are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating it.

An attacker can use cross site scripting to send malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

XSS attacks can generally be categorized into two categories: stored and reflected. Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. The browser then executes the code because it came from a 'trusted' server.

The consequence of an XSS attack is the same regardless of whether it is stored or reflected. The difference is in how the payload arrives at the server. Do not be fooled into thinking that a "read only" or "brochureware" site is not vulnerable to serious reflected XSS attacks. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user's session cookie, allowing an attacker to hijack the user's session and take over the account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, and modifying presentation of content. An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company's stock price or lessen consumer confidence. An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose.

Attackers frequently use a variety of methods to encode the malicious portion of the tag, such as using Unicode, so the request is less suspicious looking to the user. There are hundreds of variants of these attacks, including versions that do not even require any < > symbols. For this reason, attempting to "filter out" these scripts is not likely to succeed. Instead we recommend validating input against a rigorous positive specification of what is expected. XSS attacks usually come in the form of embedded JavaScript. However, any embedded active content is a potential source of danger, including: ActiveX (OLE), VBscript, Shockwave, Flash and more.

XSS issues can also be present in the underlying web and application servers as well. Most web and application servers generate simple web pages to display in the case of various errors, such as a 404 'page not found' or a 500 'internal server error.' If these pages reflect back any information from the user's request, such as the URL they were trying to access, they may be vulnerable to a reflected XSS attack.

The likelihood that a site contains XSS vulnerabilities is extremely high. There are a wide variety of ways to trick web applications into relaying malicious scripts. Developers that attempt to filter out the malicious parts of these requests are very likely to overlook possible attacks or encodings. Finding these flaws is not tremendously difficult for attackers, as all they need is a browser and some time. There are numerous free tools available that help hackers find these flaws as well as carefully craft and inject XSS attacks into a target site.

A4.2 Environments Affected

All web servers, application servers, and web application environments are susceptible to cross site scripting.

A4.3 Examples and References

- The Cross Site Scripting FAQ: http://www.cgisecurity.com/articles/xss-faq.shtml
- CERT Advisory on Malicious HTML Tags: http://www.cert.org/advisories/CA-2000-02.html
- CERT "Understanding Malicious Content Mitigation" http://www.cert.org/tech_tips/malicious_code_mitigation.html
- Cross-Site Scripting Security Exposure Executive Summary: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/topics/ExSumCS.asp
- Understanding the cause and effect of CSS Vulnerabilities: http://www.technicalinfo.net/papers/CSS.html
- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: Data Validation http://www.owasp.org/documentation/guide/
- How to Build an HTTP Request Validation Engine (J2EE validation with Stinger) http://www.owasp.org/columns/jeffwilliams/jeffwilliams2
- Have Your Cake and Eat it Too (.NET validation) http://www.owasp.org/columns/jpoteet/jpoteet2

A4.4 How to Determine If You Are Vulnerable

XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output. Note that a variety of different HTML tags can be used to transmit a malicious JavaScript. Nessus, Nikto, and some other available tools can help scan a website for these flaws, but can only scratch the surface. If one part of a website is vulnerable, there is a high likelihood that there are other problems as well.

A4.5 How to Protect Yourself

The best way to protect a web application from XSS attacks is ensure that your application performs validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed. The validation should not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content. We strongly recommend a 'positive' security policy that specifies what is allowed. 'Negative' or attack signature based policies are difficult to maintain and are likely to be incomplete.

Encoding user supplied output can also defeat XSS vulnerabilities by preventing inserted scripts from being transmitted to users in an executable form. Applications can gain significant protection from javascript based attacks by converting the following characters in all generated output to the appropriate HTML entity encoding:

From:	To:
<	&lt;
>	&gt;
(	&#40;
)	&#41;
#	&#35;
&	&#38;

The OWASP Filters project is producing reusable components in several languages to help prevent many forms of parameter tampering, including the injection of XSS attacks. OWASP has also released CodeSeeker, an application level firewall. In addition, the OWASP WebGoat training program has lessons on Cross Site Scripting and data encoding.

A5 Buffer Overflows

A5.1 Description

Attackers use buffer overflows to corrupt the execution stack of a web application. By sending carefully crafted input to a web application, an attacker can cause the web application to execute arbitrary code – effectively taking over the machine. Buffer overflows are not easy to discover and even when one is discovered, it is generally extremely difficult to exploit. Nevertheless, attackers have managed to identify buffer overflows in a staggering array of products and components. Another very similar class of flaws is known as format string attacks.

Buffer overflow flaws can be present in both the web server or application server products that serve the static and dynamic aspects of the site, or the web application itself. Buffer overflows found in widely used server products are likely to become widely known and can pose a significant risk to users of these products. When web applications use libraries, such as a graphics library to generate images, they open themselves to potential buffer overflow attacks.

Buffer overflows can also be found in custom web application code, and may even be more likely given the lack of scrutiny that web applications typically go through. Buffer overflow flaws in custom web applications are less likely to be detected because there will normally be far fewer hackers trying to find and exploit such flaws in a specific application. If discovered in a custom application, the ability to exploit the flaw (other than to crash the application) is significantly reduced by the fact that the source code and detailed error messages for the application are normally not available to the hacker.

A5.2 Environments Affected

Almost all known web servers, application servers, and web application environments are susceptible to buffer overflows, the notable exception being Java and J2EE environments, which are immune to these attacks (except for overflows in the JVM itself).

A5.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: Data Validation http://www.owasp.org/documentation/guide/
- Aleph One, “Smashing the Stack for Fun and Profit”, http://www.phrack.com/show.php?p=49&a=14
- Mark Donaldson, “Inside the Buffer Overflow Attack: Mechanism, Method, & Prevention”, http://rr.sans.org/code/inside_buffer.php

A5.4 How to Determine If You Are Vulnerable

For server products and libraries, keep up with the latest bug reports for the products you are using. For custom application software, all code that accepts input from users via the HTTP request must be reviewed to ensure that it can properly handle arbitrarily large input.

A5.5 How to Protect Yourself

Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure. Apply the latest patches to these products. Periodically scan your web site with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications.

For your custom application code, you need to review all code that accepts input from users via the HTTP request and ensure that it provides appropriate size checking on all such inputs. This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.

A6 Injection Flaws

A6.1 Description

Injection flaws allow attackers to relay malicious code through a web application to another system. These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL (i.e., SQL injection). Whole scripts written in perl, python, and other languages can be injected into poorly designed web applications and executed. Any time a web application uses an interpreter of any type there is a danger of an injection attack.

Many web applications use operating system features and external programs to perform their functions. Sendmail is probably the most frequently invoked external program, but many other programs are used as well. When a web application passes information from an HTTP request through as part of an external request, it must be carefully scrubbed. Otherwise, the attacker can inject special (meta) characters, malicious commands, or command modifiers into the information and the web application will blindly pass these on to the external system for execution.

SQL injection is a particularly widespread and dangerous form of injection. To exploit a SQL injection flaw, the attacker must find a parameter that the web application passes through to a database. By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database. These attacks are not difficult to attempt and more tools are emerging that scan for these flaws. The consequences are particularly damaging, as an attacker can obtain, corrupt, or destroy database contents.

Injection attacks can be very easy to discover and exploit, but they can also be extremely obscure. The consequences can also run the entire range of severity, from trivial to complete system compromise or destruction. In any case, the use of external calls is quite widespread, so the likelihood of a web application having a command injection flaw should be considered high.

A6.2 Environments Affected

Every web application environment allows the execution of external commands such as system calls, shell commands, and SQL requests. The susceptibility of an external call to command injection depends on how the call is made and the specific component that is being called, but almost all external calls can be attacked if the web application is not properly coded.

A6.3 Examples and References

- Examples: A malicious parameter could modify the actions taken by a system call that normally retrieves the current user's file to access another user's file (e.g., by including path traversal "../" characters as part of a filename request). Additional commands could be tacked on to the end of a parameter that is passed to a shell script to execute an additional shell command (e.g., "; rm –r *") along with the intended command. SQL queries could be modified by adding additional 'constraints' to a where clause (e.g., "OR 1=1") to gain access to or modify unauthorized data.
- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: Data Validation http://www.owasp.org/documentation/guide/
- How to Build an HTTP Request Validation Engine (J2EE validation with Stinger) http://www.owasp.org/columns/jeffwilliams/jeffwilliams2
- Have Your Cake and Eat it Too (.NET validation) http://www.owasp.org/columns/jpoteet/jpoteet2
- White Paper on SQL Injection: http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

A6.4 How to Determine If You Are Vulnerable

The best way to determine if you are vulnerable to command injection attacks is to search the source code for all calls to external resources (e.g., system, exec, fork, Runtime.exec, SQL queries, or whatever the syntax is for making requests to

interpreters in your environment). Note that many languages have multiple ways to run external commands. Developers should review their code and search for all places where input from an HTTP request could possibly make its way into any of these calls. You should carefully examine each of these calls to be sure that the protection steps outlined below are followed.

A6.5 How to Protect Yourself

The simplest way to protect against injection is to avoid accessing external interpreters wherever possible. For many shell commands and some system calls, there are language specific libraries that perform the same functions. Using such libraries does not involve the operating system shell interpreter, and therefore avoids a large number of problems with shell commands.

For those calls that you must still employ, such as calls to backend databases, you must carefully validate the data provided to ensure that it does not contain any malicious content. You can also structure many requests in a manner that ensures that all supplied parameters are treated as data, rather than potentially executable content. The use of stored procedures or prepared statements will provide significant protection, ensuring that supplied input is treated as data. These measures will reduce, but not completely eliminate the risk involved in these external calls. You still must always validate such input to make sure it meets the expectations of the application in question.

Another strong protection against command injection is to ensure that the web application runs with only the privileges it absolutely needs to perform its function. So you should not run the webserver as root or access a database as DBADMIN, otherwise an attacker can abuse these administrative privileges granted to the web application. Some of the J2EE environments allow the use of the Java sandbox, which can prevent the execution of system commands.

If an external command must be used, any user information that is being inserted into the command should be rigorously checked. Mechanisms should be put in place to handle any possible errors, timeouts, or blockages during the call.

All output, return codes and error codes from the call should be checked to ensure that the expected processing actually occurred. At a minimum, this will allow you to determine that something has gone wrong. Otherwise, the attack may occur and never be detected.

The OWASP Filters project is producing reusable components in several languages to help prevent many forms of injection. OWASP has also released CodeSeeker, an application level firewall.

A7 Improper Error Handling

A7.1 Description

Improper handling of errors can introduce a variety of security problems for a web site. The most common problem is when detailed internal error messages such as stack traces, database dumps, and error codes are displayed to the user (hacker). These messages reveal implementation details that should never be revealed. Such details can provide hackers important clues on potential flaws in the site and such messages are also disturbing to normal users.

Web applications frequently generate error conditions during normal operation. Out of memory, null pointer exceptions, system call failure, database unavailable, network timeout, and hundreds of other common conditions can cause errors to be generated. These errors must be handled according to a well thought out scheme that will provide a meaningful error message to the user, diagnostic information to the site maintainers, and no useful information to an attacker.

Even when error messages don't provide a lot of detail, inconsistencies in such messages can still reveal important clues on how a site works, and what information is present under the covers. For example, when a user tries to access a file that does not exist, the error message typically indicates, "file not found". When accessing a file that the user is not authorized for, it indicates, "access denied". The user is not supposed to know the file even exists, but such inconsistencies will readily reveal the presence or absence of inaccessible files or the site's directory structure.

One common security problem caused by improper error handling is the fail-open security check. All security mechanisms should deny access until specifically granted, not grant access until denied, which is a common reason why fail open errors occur. Other errors can cause the system to crash or consume significant resources, effectively denying or reducing service to legitimate users.

Good error handling mechanisms should be able to handle any feasible set of inputs, while enforcing proper security. Simple error messages should be produced and logged so that their cause, whether an error in the site or a hacking attempt, can be reviewed. Error handling should not focus solely on input provided by the user, but should also include any errors that can be generated by internal components such as system calls, database queries, or any other internal functions.

A7.2 Environments Affected

All web servers, application servers, and web application environments are susceptible to error handling problems.

A7.3 Examples and References

- OWASP discussion on generation of error codes: http://www.owasp.org/documentation/guide/

A7.4 How to Determine If You Are Vulnerable

Typically, simple testing can determine how your site responds to various kinds of input errors. More thorough testing is usually required to cause internal errors to occur and see how the site behaves.

Another valuable approach is to have a detailed code review that searches the code for error handling logic. Error handling should be consistent across the entire site and each piece should be a part of a well-designed scheme. A code review will reveal how the system is intended to handle various types of errors. If you find that there is no organization to the error-handling scheme or that there appear to be several different schemes, there is quite likely a problem.

A7.5 How to Protect Yourself

A specific policy for how to handle errors should be documented, including the types of errors to be handled and for each, what information is going to be reported back to the user, and what information is going to be logged. All developers need to understand the policy and ensure that their code follows it.

In the implementation, ensure that the site is built to gracefully handle all possible errors. When errors occur, the site should respond with a specifically designed result that is helpful to the user without revealing unnecessary internal details. Certain classes of errors should be logged to help detect implementation flaws in the site and/or hacking attempts.

Very few sites have any intrusion detection capabilities in their web application, but it is certainly conceivable that a web application could track repeated failed attempts and generate alerts. Note that the vast majority of web application attacks are never detected because so few sites have the capability to detect them. Therefore, the prevalence of web application security attacks is likely to be seriously underestimated.

The OWASP Filters project is producing reusable components in several languages to help prevent error codes leaking into user's web pages by filtering pages when they are constructed dynamically by the application.

A8 Insecure Storage

A8.1 Description

Most web applications have a need to store sensitive information, either in a database or on a file system somewhere. The information might be passwords, credit card numbers, account records, or proprietary information. Frequently, encryption techniques are used to protect this sensitive information. While encryption has become relatively easy to implement and use, developers still frequently make mistakes while integrating it into a web application. Developers may overestimate the protection gained by using encryption and not be as careful in securing other aspects of the site. A few areas where mistakes are commonly made include:

- Failure to encrypt critical data
- Insecure storage of keys, certificates, and passwords
- Improper storage of secrets in memory
- Poor sources of randomness
- Poor choice of algorithm
- Attempting to invent a new encryption algorithm
- Failure to include support for encryption key changes and other required maintenance procedures

The impact of these weaknesses can be devastating to the security of a website. Encryption is generally used to protect a site's most sensitive assets, which may be totally compromised by a weakness.

A8.2 Environments Affected

Most web application environments include some form of cryptographic support. In the rare case that such support is not already available, there are a wide variety of third-party products that can be added. Only web sites that use encryption to protect information in storage or transit are susceptible to these attacks. Note that this section does not cover the use of SSL, which is covered in A10 Insecure Configuration Management. This section deals only with programmatic encryption of application layer data.

A8.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services
  http://www.owasp.org/documentation/guide/
- Bruce Schneier, "Applied Cryptography", 2nd edition, John Wiley & Sons, 1995

A8.4 How to Determine If You Are Vulnerable

Discovering cryptographic flaws without access to the source code can be extremely time consuming. However, it is possible to examine tokens, session IDs, cookies and other credentials to see if they are obviously not random. All the traditional cryptanalysis approaches can be used to attempt to uncover how a web site is using cryptographic functions.

By far the easiest approach is to review the code to see how the cryptographic functions are implemented. A careful review of the structure, quality, and implementation of the cryptographic modules should be performed. The reviewer should have a strong background in the use of cryptography and common flaws. The review should also cover how keys, passwords, and other secrets are stored, protected, loaded, processed, and cleared from memory.

A8.5 How to Protect Yourself

The easiest way to protect against cryptographic flaws is to minimize the use of encryption and only keep information that is absolutely necessary. For example, rather than encrypting credit card numbers and storing them, simply require users to re-enter the numbers. Also, instead of storing encrypted passwords, use a one-way function, such as SHA-1, to hash the passwords.

If cryptography must be used, choose a library that has been exposed to public scrutiny and make sure that there are no open vulnerabilities. Encapsulate the cryptographic functions that are used and review the code carefully. Be sure that secrets, such as keys, certificates, and passwords, are stored securely. To make it difficult for an attacker, the master secret should be split into at least two locations and assembled at runtime. Such locations might include a configuration file, an external server, or within the code itself.

A9 Denial of Service

A9.1 Description

Web applications are particularly susceptible to denial of service attacks. Note that network denial of service attacks, such as SYN floods, are a separate problem that is outside the scope of this document.

A web application can't easily tell the difference between an attack and ordinary traffic. There are many factors that contribute to this difficulty, but one of the most important is that, for a number of reasons, IP addresses are not useful as an identification credential. Because there is no reliable way to tell where an HTTP request is from, it is very difficult to filter out malicious traffic. For distributed attacks, how would an application tell the difference between a true attack, multiple users all hitting reload at the same time (which might happen if there is a temporary problem with the site), or getting "slashdotted"?

Most web servers can handle several hundred concurrent users under normal use. A single attacker can generate enough traffic from a single host to swamp many applications. Load balancing can make these attacks more difficult, but far from impossible, especially if sessions are tied to a particular server. This is a good reason to make an application's session data as small as possible and to make it somewhat difficult to start a new session.

Once an attacker can consume all of some required resource, they can prevent legitimate users from using the system. Some resources that are limited include bandwidth, database connections, disk storage, CPU, memory, threads, or application specific resources. All of these resources can be consumed by attacks that target them. For example, a site that allows unauthenticated users to request message board traffic may start many database queries for each HTTP request they receive. An attacker can easily send so many requests that the database connection pool will get used up and there will be none left to service legitimate users.

Other variants of these attacks target system resources related to a particular user. For example, an attacker might be able to lock out a legitimate user by sending invalid credentials until the system locks out the account. Or the attacker might request a new password for a user, forcing them to access their email account to regain access. Alternatively, if the system locks any resources for a single user, then an attacker could potentially tie them up so others could not use them.

Some web applications are even susceptible to attacks that will take them offline immediately. Applications that do not properly handle errors can even take down the web application container. These attacks are particularly devastating because they instantly prevent all other users from using the application.

There are a wide variety of these attacks, most of which can be easily launched with a few lines of perl code from a low powered computer. While there are no perfect defenses to these attacks, it is possible to make it more difficult for them to succeed.

A9.2 Environments Affected

All web servers, application servers, and web application environments are susceptible to denial of service attacks.

A9.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services
  http://www.owasp.org/documentation/guide/

A9.4 How to Determine If You Are Vulnerable

One of the hardest parts of denial of service attacks is determining whether you are vulnerable. Load testing tools, such as JMeter can generate web traffic so that you can test certain aspects of how your site performs under heavy load. Certainly one important test is how many requests per second your application can field. Testing from a single IP address is useful as it will give you an idea of how many requests an attacker will have to generate in order to damage your site.

To determine if any resources can be used to create a denial of service, you should analyze each one to see if there is a way to exhaust it. You should particularly focus on what an unauthenticated user can do, but unless you trust all of your users, you should examine what an authenticated user can do as well.

A9.5 How to Protect Yourself

Defending against denial of service attacks is difficult, as there is no way to protect against these attacks perfectly.

As a general rule, you should limit the resources allocated to any user to a bare minimum. For authenticated users, it is possible to establish quotas so that you can limit the amount of load a particular user can put on your system. In particular, you might consider only handling one request per user at a time by synchronizing on the user's session. You might also consider dropping any requests that you are currently processing for a user when another request from that user arrives.

For unauthenticated users, you should avoid any unnecessary access to databases or other expensive resources. Try to architect the flow of your site so that an unauthenticated user will not be able to invoke any expensive operations. You might consider caching the content received by unauthenticated users instead of generating it or accessing databases to retrieve it.

You should also check your error handling scheme to ensure that an error cannot affect the overall operation of the application.

A10 Insecure Configuration Management

A10.1 Description

Web server and application server configurations play a key role in the security of a web application. These servers are responsible for serving content and invoking applications that generate content. In addition, many application servers provide a number of services that web applications can use, including data storage, directory services, mail, messaging, and more. Failure to manage the proper configuration of your servers can lead to a wide variety of security problems.

Frequently, the web development group is separate from the group operating the site. In fact, there is often a wide gap between those who write the application and those responsible for the operations environment. Web application security concerns often span this gap and require members from both sides of the project to properly ensure the security of a site's application.

There are a wide variety of server configuration problems that can plague the security of a site. These include:

- Unpatched security flaws in the server software
- Server software flaws or misconfigurations that permit directory listing and directory traversal attacks
- Unnecessary default, backup, or sample files, including scripts, applications, configuration files, and web pages
- Improper file and directory permissions
- Unnecessary services enabled, including content management and remote administration
- Default accounts with their default passwords
- Administrative or debugging functions that are enabled or accessible
- Overly informative error messages (more details in the error handling section)
- Misconfigured SSL certificates and encryption settings
- Use of self-signed certificates to achieve authentication and man-in-the-middle protection
- Use of default certificates
- Improper authentication with external systems

Some of these problems can be detected with readily available security scanning tools. Once detected, these problems can be easily exploited and result in total compromise of a website. Successful attacks can also result in the compromise of backend systems including databases and corporate networks. Having secure software and a secure configuration are both required in order to have a secure site.

A10.2 Environments Affected

All web servers, application servers, and web application environments are susceptible to misconfiguration.

A10.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services
  http://www.owasp.org/documentation/guide/
- Web Server Security Best Practices: http://www.pcmag.com/article2/0,4149,11525,00.asp
- Securing Public Web Servers (from CERT): http://www.cert.org/security-improvement/modules/m11.html

A10.4 How to Determine If You Are Vulnerable

If you have not made a concerted effort to lock down your web and application servers you are most likely vulnerable. Few, if any, server products are secure out of the box. A secure configuration for your platform should be documented and updated frequently. A manual review of the configuration guide should be performed regularly to ensure that it has been kept up to date and is consistent. A comparison to the actual deployed systems is also recommended.

In addition, there are a number of scanning products available that will externally scan a web or application server for known vulnerabilities, including Nessus and Nikto. You should run these tools on a regular basis, at least monthly, to find problems as soon as possible. The tools should be run both internally and externally. External scans should be run from a host located external to the server's network. Internal scans should be run from the same network as the target servers.

A10.5 How to Protect Yourself

The first step is to create a hardening guideline for your particular web server and application server configuration. This configuration should be used on all hosts running the application and in the development environment as well. We recommend starting with any existing guidance you can find from your vendor or those available from the various existing security organizations such as OWASP, CERT, and SANS and then tailoring them for your particular needs. The hardening guideline should include the following topics:

- Configuring all security mechanisms
- Turning off all unused services
- Setting up roles, permissions, and accounts, including disabling all default accounts or changing their passwords
- Logging and alerts

Once your guideline has been established, use it to configure and maintain your servers. If you have a large number of servers to configure, consider semi-automating or completely automating the configuration process. Use an existing configuration tool or develop your own. A number of such tools already exist. You can also use disk replication tools such as Ghost to take an image of an existing hardened server, and then replicate that image to new servers. Such a process may or may not work for you given your particular environment.

Keeping the server configuration secure requires vigilance. You should be sure that the responsibility for keeping the server configuration up to date is assigned to an individual or team. The maintenance process should include:

- Monitoring the latest security vulnerabilities published
- Applying the latest security patches
- Updating the security configuration guideline
- Regular vulnerability scanning from both internal and external perspectives
- Regular internal reviews of the server's security configuration as compared to your configuration guide
- Regular status reports to upper management documenting overall security posture

Conclusions

OWASP has assembled this list to raise awareness about web application security. The experts at OWASP have concluded that these vulnerabilities represent a serious risk to agencies and companies that have exposed their business logic to the Internet. Web application security problems are as serious as network security problems, although they have traditionally received considerably less attention. Attackers have begun to focus on web application security problems, and are actively developing tools and techniques for detecting and exploiting them.

This Top Ten list is only a starting point. We believe that these flaws represent the most serious risks to web application security, but there are many other security critical areas that were considered for the list and also represent significant risk to organizations deploying web applications. These include flaws in the areas of:

- Unnecessary and Malicious Code
- Broken Thread Safety and Concurrent Programming
- Unauthorized Information Gathering
- Accountability Problems and Weak Logging
- Data Corruption
- Broken Caching, Pooling, and Reuse

We welcome your feedback on this Top Ten list. Please participate in the OWASP mailing lists and help to improve web application security. Visit http://www.owasp.org to get started.