Secure Web Application Coding Team  
January 26, 2006  
1:00 – 2:00PM  
Bits & Pieces Room, Sansom West Room 306

Agenda

1. Dan – Please take minutes?
2. Team Website –
   a. Content areas:
      i. Meeting Agendas & Meeting Minutes
      ii. Tasks - Who is working on what
      iii. Work In Progress - documents in draft
      iv. Web App Security Team Deliverables - these would be pdfs
      v. Miscellaneous

3. Follow up Q: Confirm basic goal to update OWASP to fit Penn/UPHS
   a. “The Open Web Application Security Project (OWASP) is an all-volunteer group that produces free, professional-quality, open-source documentation, tools, and standards.”

   May 13th, 2005 - First Meeting
   OWASP's Philadelphia first meeting will cover an introduction to web application security.
   Time: 12:00 – 2:00
   Agenda:
   Lunch
   OWASP Introduction – Aaron Weaver
   Innocent Code "rules" – Shawn Smolsky
   Q&A

   Please RSVP to: aweaver@sovereignbank.com
   Location: 1125 Berkshire Blvd. Wyomissing, PA
   Directions: From Philadelphia area take 422 West. Take the exit toward Lebanon/422 West and exit at the Papermill exit. Go directly through two lights and take a left on Berkshire Blvd. Go through the first light and the building is on the right directly across from the green glass Sovereign Bank building.

5. Discuss the tasks/updates team members.
6. OWASP #1/2– I entered the Team Additions/Updates highlighted in Blue within OWASP wording below.
7. OWASP #3 – Broken Authentication and Session Management (30 minutes)
8. OWASP #4 – Cross Site Scripting (XSS) Flaws (time permitting)
Current OWASP Wording:
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.

Susan to draft wording on risks here

A1.2 Environments Affected

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

The following grid lists some of the available tools for checking for unvalidated input across different development platforms.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Available Tools</th>
</tr>
</thead>
</table>
| PHP      | **Safe Mode** – Protects against individual users setting dangerous configuration values.  
**Error Reporting** – Setting error reporting to E_ALL will warn against using variables that have not been initialized or checked.  
**Register Globals** – As of PHP 4.2.0, the default value for register_globals is off. This prevents the injection of values into variables through querystring or form values through global registration. |
**ASP.NET**

*ValidateRequest Page Directive* – By default, the ValidateRequest page directive is set to true. This causes an exception to be thrown if user input matches a list of potentially dangerous values. This protects against many XSS type attacks.

*Strong Typing* – Variables used in the ASP.NET page or in code-behind files are strongly typed. Explicit casting can be enforced by using the Strict option in VB.NET and is always enforced in other .NET languages.

**J2EE**

*Stinger* – Stinger is an HTTP Validation Engine developed by OWASP. It checks input against a configurable set of potential threats. This product may become available for other development environments.

*Strong Typing* – Java is a strongly typed language, requiring stricter coding than in a loosely typed language.

**Perl**

*Taint* – Taint marks variables as “tainted” during program operation and causes a taint violation if the tainted data is used in operations which could be potentially dangerous.

**Cold Fusion**

*Input from Terry?*

### A1.3 Examples and References

- OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: Data Validation
- modsecurity project (Apache module for HTTP validation) [http://www.modsecurity.org](http://www.modsecurity.org)
- How to Build an HTTP Request Validation Engine (J2EE validation with Stinger)
  [http://www.owasp.org/columns/jeffwilliams/jeffwilliams2](http://www.owasp.org/columns/jeffwilliams/jeffwilliams2)
- Have Your Cake and Eat it Too (.NET validation)
  [http://www.owasp.org/columns/poteet/poteet2](http://www.owasp.org/columns/poteet/poteet2)

### 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.

Specific examples include:
- Global Variables Exploit w/Post and Get – initialize global variables
- Cross-site Scripting – this is a universal attack, script tags is a well known function
- SQL Injection – use binding, if not use strings or integers (put integers in quotes)
- Group requested to add other examples ---

### 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.

Everyone to draft examples here

**OWASP #2 -**
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.

PENN/UPHS

- UPHS: For Medview NT Authorization is used.
- Use PennKey, where possible. We should identify instance for authorization outside of Penn for those that don’t have a PennKey.
- Websec is good but many are not using a logout mechanism to destroy tokens (the risk is someone could take your token and login to your session again). Guidelines for PennKey system should be created.
- Some servers are running with higher privileges than needed.

A2.3 Examples and References

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 sound 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**


**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 referer 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.
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
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:
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:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>&amp;lt;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>&amp;gt;</code></td>
</tr>
<tr>
<td>`(</td>
<td><code>&amp;#40;</code></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>&amp;#41;</code></td>
</tr>
<tr>
<td><code>#</code></td>
<td><code>&amp;#35;</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>&amp;#38;</code></td>
</tr>
</tbody>
</table>

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.