Countering Cross-Site Scripting in Web-based Applications

Loye Lynn Ray
Cyber Security and Information Assurance
University of Maryland University College
3501 University Blvd East, Adelphi, MD 20783
love.ray@faculty.umuc.edu
Phone 443-729-1045

Abstract: Today’s dynamic web-based applications have become a normal and critical asset to an organization’s business. They come with an increase in the number of web vulnerabilities and attacks. These weaknesses allow hackers to focus their attention on attacking this important information source. The most common vulnerability is cross-site scripting (XSS) and one of the Open Web Application Security project (OWASP) top ten web-threats. XSS occurs when a Web-based application allows untrusted information be accepted and sent back to a browser. Also they can execute scripts within a browser that can deface web sites, redirect users to malicious content and hijack browsers. One reason for this problem was the lack of developers understanding the causes of XSS. In this paper, we address the causes of XSS and countermeasures to defense against these threats.

Keywords: Cross-Site Scripting, Reflected XSS, Stored XSS, Input Validation, Web Proxy, Scripting

INTRODUCTION

Web-based applications play a critical part in our lives and a way to accessing information globally. They provide a vital communications channel between service providers and Internet users (Jovanovic, Kregel & Kirda, 2010). This emerging strategy and change in our lives has set a new paradigm in the way we treat information. These applications provide access to information for millions of people around the world. Improvement to Web-based technology with such things as mobile apps has opened up new avenues and software techniques for developers. Using AJAX, Java Script and other languages help make this possible.

However, inclusion of effective security mechanisms to these applications is a growing concern (Jovanovic, Kregel & Kirda, 2010; Garcia-Alfaro & Navarro-Arribas, 2009). The increase in value to these applications offer an organization reliable security mechanisms to protect user data. Unfortunately, this increase in technology has opened the door to new vulnerabilities and threats. These vulnerabilities result into increasing compromises of sensitive information leading to damages of an organization or personal nature (Selvamani, Duraisamy & Kannan, 2010). These applications are frequently targeted by attackers and lead to compromises in sensitive corporate information. It can also cause lost of a company’s reputation, financial loss and lawsuits. Thus, web application security plays a big part of its normal operations (Steinke, Tunrea & Kelly, 2011).

Hackers now have focused their attention to attacking this important information source. This method is called cross-site scripting (XSS) and can contaminate a Web site or any user’s browser
accessing the information on it. It has become one of the leading threats to today’s applications (Selvamani, Duraisamy & Kannan, 2010). This type of attack happens when a web-based application is infected with a malicious code such as JavaScript that can be called from a user’s browser (Rao, Tejaswini & Preethi, 2012; Bisht & Venkatakrishnan, 2008). It can also use code injection from other languages such as Java, Flash, HTML and VBScript. This can complicate the process of detecting and preventing XSS because of the different way browser’s interpret the code (Saha, 2009). Thus, an XSS attack can be a serious threat to any organization doing business on the Web. XSS can cause users to lose respect for a Web site. It has become one of the biggest problems for developers of web applications. Fixing these vulnerabilities in a large system can be challenging and may not be accomplished (Bates, Barth & Jackson, 2010). Even worst, businesses could close down due to loss of revenue. Combating this threat is necessary to preserve the freedom of using the Web and Internet. This paper is divided up into two sections to describe XSS and ways to overcome this threat. The first section explains what XSS is, how it works and different categories. The last section provides detailed countermeasures one can deploy to combat the threat. The result is an understanding what is needed to protect Web sites from this malicious attack.

OVERVIEW OF CROSS-SITE SCRIPTING

Before countering XSS, one needs to understand how they work in detail. Understanding the weaknesses of Web applications and what methods attackers use will be important to combating these threats.

What is it?

According to OWASP (2013) and Harris (2010), cross-site scripting is an attack that exploits a vulnerability that allows malicious code to be placed in a Web application. Attackers send a malicious code based on a browser script to an unsuspecting user. The data is added to dynamic content and sent to the user without being checked for malicious code or validity. The user’s browser activates the malicious code by clicking on a Uniform Resource Locator (URL) or other content being displayed. The attacker can use this attack to create phishing sites and capture Web session for stealing sensitive data (Wurzinger, Platzer, Ludl, Kirda & Kruegel, 2009). This malicious code can also hijack user sessions or get cookies, tokens and other information about users (Steinke, Tundrea & Kelly, 2011; Garcia-Alfaro & Navarro-Arribas, 2009). OWASP (2011) also mentions XSS attackers can use scripts to change Web site content in a Hyper Text Transfer Language (HTML) page.

Cross-site scripting vulnerabilities are considered design and coding errors because of a failure to handle inputs to the Web-based application (Steinke, Tundrea & Kelly, 2011). They can also use the developer’s inability to check input and output code to attack a Web application on the server. This incomplete or incorrect input sanitization technique makes web-based application vulnerable to XSS (Wang, Mao & Lee, 2010; Saha, 2009). Through this hole they can inject scripts written in JavaScript to steal a victims sensitive data or Web session. XSS can also capture a user’s session cookie, hijacking their session, install malicious code such as a Trojan horse or redirect users to another Web site. Cookies provide the authentication between the
application and user (Gupta, Sharma, Gupta & Gupta, 2012). When accessed by a Web browser, the user’s computer gets infected. Even fixing the server doesn’t help the user.

**How does it work?**

In today’s Web-based applications, XSS attackers try to steal information from the Web browser’s database (Guota, Sharma, Gupta & Gupta, 2012). This involves stealing the cookies that contain the user’s authentication information or credentials. The following steps help explain the process of a generic XSS attack.

First, the attacker will find a vulnerable web application to exploit. Then they will post a malicious script such as JavaScript on the vulnerable server. The function of the script is to steal a user’s cookie containing the session account information (Guota, Sharma, Gupta & Gupta, 2012). Second, the user logs into the application with their account information to the infected server not knowing it has been infected. Third, the victim is directed to the malicious code where it is posted and executed on their browser (Guota, Sharma, Gupta & Gupta, 2012). Next, the interpreter in the victim’s browser invokes a transfer of the cookie containing the account information for that session. Lastly, the attacker can use these cookies to access the victim’s account.

When a user sends a web request it includes a cookie with authentication information of the user. The server receiving the cookie treat it as a request form the user until they log out. If an attacker can obtain the cookie, they can impersonate an authenticated user during the current session (Jovanovic, Kregel & Kirda, 2010). A user’s browser only sends a cookie to the web application that created it. However, using malicious JavaScript, one can send the cookie to anywhere. Attackers use the JavaScript access rights that restrict it to a same-origin policy. This means that the JavaScript can only access cookies belonging to the application or server the code came from (Jovanovic, Kregel & Kirda, 2010). XSS attacks get around this policy by injecting malicious code to vulnerable web-based applications. The malicious code will seem to come from the trusted application and have complete access to a user’s sensitive information (Jovanovic, Kregel & Kirda, 2010).

One way of understanding how XSS works is through examples. OWASP (2011) provides some examples of XSS using scripts within attributes such as:

<body onload=alert(‘execute2’)>   (1)

or

<b onmouseover=alert(‘Meow!’)>click here!</b>   (2)

or

<img src=http://url.bad.file/that.not.here onerror=alert(document.cookie);>  (3)

or


These examples illustrate active content embedded within a Web reply that can get information about users or cause them to accept the tainted messages. More advanced scripts can be constructed to perform more severe damage. Other examples include where the malicious code can input a string using a variable value:

```
“ ”<script> attack() </script>”
```

The double quote triggers the vulnerability. Also this example is better since it can be embedded in a web page on the server.

```
<script>attack</script> <!--a name=""
```

Still more examples include using more variances of “>” with or without an ending tag such as `</p>` include:

```
“>
“> ‘> </p>
>”> </p>  
“> </p>
```

Where `</p>` can be also `</div>`, `<sCrIpT>` or `<form>` (Wang, Mao & Lee, 2010).

The challenge is to implement techniques and mechanisms that can automatically detect and stop these types of attacks. Thus, learning the different structures of attack vectors used in XSS, one can apply the right method of preventing them. This also includes finding the right testing tool to accomplish this. In some cases one might consider using sanitization methods such as blacklist or whitelist (Wang, Mao & Lee, 2010).

**Taint-based vulnerabilities**

XSS is an instance of taint vulnerabilities and its purpose is to steal users credentials (Jovanovic, Kregel & Kirda, 2010). These vulnerabilities require the attacker to inject their malicious code into a web-based application. Then it can be exploited in different ways including the browser (Martin & Lam, 2011). An example of these vulnerabilities is using reflected XSS to steal a user’s credentials. This is accomplished by injecting malicious code into a web-based application site. A problem with detecting these vulnerabilities is that the stolen credentials are valid and whitelisting can’t stop the attack since the application site is also trusted (Martin & Lam, 2011).

**Categories of XSS Attacks**
There are basically three categories of XSS attacks. These include Persistent, Non-persistent and Local-based XSS. Each is described below.

**Persistent XSS Attacks**

The persistent or stored XSS attack is continuously the most powerful of the different attacks (Tang, Zhu, Cao & Zhao, 2011). The attacker sends malicious code to the web server stored on a server. This code contains attack vectors that can reside in a variety of places to include database, file or any location on the server. When a browser is used to request information from the server, the malicious attack vectors are displayed on the users web browser (Gilad & Herzberg, 2012; Guota, Sharma, Gupta & Gupta, 2012). Attackers use this vulnerability to access the users browser for stealing their information such as passwords and email addresses. This type of attack is more significant than other types because the malicious code automatically infects the users system without having to redirect them to another site. Also the attackers can control data from the web application (Rao, Tejaswini & Preethi, 2012). The script isn’t removed from the messages but allows the attacker to obtain sensitive information. A good example is one that convinces you to click on a link because you have won something. A script is embedded that redirects you to a malicious site and steals your cookie. An example of a persistent XSS script may look like (Saha, 2009):

```html
</script>
```

Persistent XSS attacks also can use applications that don’t utilize strong input validation mechanisms (Garcia-Alfaro & Navarro-Arribas, 2009). This weakness could allow hackers to inject malicious JavaScript code into the Web application. A user then accesses the application as a trusted context not realizing it contains the malicious code. Having a little or no input validation checking mechanism, allows the application to store the code in the data repository. From here, the browser allows the malicious code to access the location of users’ cookies (Garcia-Alfaro & Navarro-Arribas, 2009). Then it can steal someone’s sensitive information by circumventing security policies restricting data access by JavaScript. These policies restrict scripts to only be allowed if they originate from the same place as where the data is located (Garcia-Alfaro & Navarro-Arribas, 2009). This is because the malicious code resides in the same location as the data.

A malicious JavaScript code can also use a false login form to simulate a user logout (Garcia-Alfaro & Navarro-Arribas, 2009). This form can store a victims login information such as userID, password, etc. It can even get one’s secret questions and answers allowing the attacker to better impersonate the victim. Attackers can use this information to make legitimate logins to the application (Garcia-Alfaro & Navarro-Arribas, 2009).

Like the non-persistent, it too exploits a bug in the software application. This bug prevents the use of network security devices such as firewalls and intrusion prevention systems from being effective against persistent XSS attacks (Tang, Zhu, Cao & Zhao, 2011).

**Non-persistent XSS Attacks**
This is the most common XSS vulnerability and attack (Selvamani, Duraisamy & Kannan, 2010; Saha, 2009). Non-persistent or reflected XSS attack is where the malicious code is reflected off the server as an error message that includes part of the input message (OWASP, 2011). The XSS code is sent to a user via an email or from another Web server. The idea is that the user is tricked into clicking on a link and the malicious code is sent to a Web server. A sample non-persistent XSS code that an attacker can use to get a victim to click on a malicious link may look like (Saha, 2009):

\[
\text{\textless a href=\textquote{http://www.goodsite.com/}} \\
\text{\textless ?script\textgreater location=\textquote{http://www.malicious-site.com/stealcookie.php\?a.cookie}; (7) \\
\text{\textless \textquoteright script\textquoteright} Click here for a free gift \textless /a\textgreater
\]

Then it is reflected back to the user where their browser executes the code thinking it came from a trusted source (Gilad & Herzberg, 2012; Gupta, Sharma, Guota & Gupta, 2012; Jovanovic, Kregel & Kirda, 2010). This makes it different from persistent XSS attacks because the malicious code is not stored. Email can easily be used to send the victim a link that contains the malicious code. The attacker may also use malicious JavaScript code which contain a query string to obtain the attack code (Saha, 2009). Spoofed emails have been used as a transport mechanism for getting the link to the malicious code to a victim. The malicious code is then sent to the user from the trusted application site (Garcia-Alfaro & Navarro-Arribas, 2009). If the users browser doesn’t filter or block HTML control characters, this type of XSS attack can succeed. It would have to detect both HTML scripts using \textless script\textgreater or \%3cscript\%3e.

These attacks are combined with other types such as phishing. Both phishing and non-persistent XSS attacks are similar objectives in that they both attempt to steal one’s sensitive information (Garcia-Alfaro & Navarro-Arribas, 2009). The attack gets its name from the fact that the malicious code does not reside in the application site. The complexity of accomplishing this type of attack requires skilled attackers to perform the task. This type of attack works by exploiting a bug that is in the application itself.

\textit{Local XSS Attacks}

Local or Document Object Model (DOM) based XSS attacks is a standard model that uses HTML or XML contents (Rao, Tejaswini & Preethi, 2012). This type of vulnerability came from the use of Web 2.0 technology. In a DOM-based XSS attack, the browser has a vulnerability that causes it to see a script from an attacking computer as if it was coming from another server. Therefore, this attack requires one to have a bug in the browser to work (Gilad & Herzberg, 2012). This type of XSS attack is accomplished by modifying the DOM area in the client. It uses the (#) sign to get the browser to see a fragment and only send the URL to the server (Saha, 2009). An example of a local XSS malicious code that one can use to do a DOM attack can look like (Saha, 2009):

\[
\text{http://www.malicous-site.com/default.html\#name=\textless script\textgreater alert('malicious')\textless /script\textgreater} \quad (8)
\]
The unique aspect of the DOM is that the sign (#) makes XSS filters unable to detect these types of attacks. So opening another Web page while infected can change the code in the first page on the client (Selvamani, Duraisamy & Kannan, 2010).

**COUNTERMEASURES**

Using just basic security recommendations as the only measure to secure web-based application is not enough (Garcia-Alfaro & Navarro-Arribas, 2009). Doing so could allow attackers to evade detection. Now that we know what a XSS attack is, we can look at different methods of fixing this threat. However, finding XSS problems can be very difficult. The challenge now is how to make these different controls work together and resolve any differences. Manual security reviews for these vulnerabilities are prone to errors, costly and time consuming (Jovanovic, Kregel & Kirda, 2010). This leads security professionals to find faster means of detecting XSS vulnerabilities to overcome these issues. Compounding this is the growing popularity and complexity of web applications (Jovanovic, Kregel & Kirda, 2010). These challenges and issues show the need for automated vulnerability tools for detecting XSS vulnerabilities.

**Finding XSS**

A contributing factor to XSS is the lack of developers testing code for the threat before deployment. Performing security checks is a best practice. Use of fuzzing tools can help if they are configured for detecting XSS codes. These tools will inject XSS codes into an application and check the output for malicious script. The L-WMxD Webmail XSS fuzzer has been used for detecting XSS attacks in webmail applications. This active defense tool that detects XSS vulnerabilities before the application is online (Guota, Sharma, Gupta & Gupta, 2012). Another tool called Ardilla does automated testing by creating SQL injection and XSS attacks. It does this by crafting malicious content and injecting it into the code through an input generator. Acunetix automated tool can be used to test XSS vulnerabilities against exploits (Guota, Sharma, Gupta & Gupta, 2012). It allows testers to focus their tests to a specific area or device without interfering with others.

A good way for organizations to find XSS vulnerabilities is by doing scans of web-based applications and the Web sites supported (Steinke, Tundrea & Kelly, 2011). There are two different types of approaches for finding vulnerabilities in Web-based applications: Black Box and White Box. White box testing use source code and can be done manually or automatic. Manual code review is time consuming that makes automatic more appealing. This is because the person has to review each line of code to find the vulnerability and can easily miss something. Automatic static code review inputs source code into a tool that searches through every different path that an attack vector can manipulate (Tang, Zhu, Cao & Zhao, 2011). This method reduces the time necessary to find vulnerabilities in the sources code. Some good white box automated vulnerability scanners are Fortify SCA or Pixy. Pixy is used to detect XSS vulnerabilities in PHP programs. It utilizes data flow analysis on the source code to detect non-reflected XSS vulnerabilities (Guota, Sharma, Gupta & Gupta, 2012). Still other tools include AppScan and Nessus for detecting XSS vulnerabilities. However these tools only identify errors
in the code during the development cycle. They can’t provide protection or detect all vulnerabilities in the source code (Saha, 2009).

Black box scanning has become the most common. Instead of searching for vulnerabilities in the source code, black box scanners inject attack vectors directly to the Web application via HTML or URL parameters (Martin & Lam, 2011; Tang, Zhu, Cao & Zhao, 2011). There are some disadvantages to using this technique. The scanning of vulnerabilities in HTML and URL limits its effectiveness to only non-persistent XSS vulnerability detection. Another is that it can have high false positives over 20% (Tang, Zhu, Cao & Zhao, 2011). Its advantage is its independency of the programming language used in the web application. Performing regular scanning and testing of all web-based applications can significantly reduce these vulnerabilities. Also they can reduce the chances of a data breach.

Detection of any vulnerability is done using a vulnerability detector and taint propagation that reads the data flowing out of the server side of the application (Kiezun, Guo, Jayarman & Ernest, 2009). This way the server side of the application can be checked and fixed before the malicious code can infect users. One problem with using these tools is being able to generate test data. Shahriar and Zulkernine (2009) devised a tool called MUTEC that automatically generates different mutation elements that can be used as a dataset. These are used for testing PHP and JavaScript environments that may be contaminated with malicious code. Other tools such as Nessus, Nikto, N-Stalker or Acunetix can also help identify XSS vulnerabilities.

Gupta, Sharma, Gupts and Gupta (2012), and Wassermann and Su (2008) devised using static analysis to find vulnerabilities within a server side application. Specifically it looks for weaknesses in input validation that is a contributor of XSS attacks. It combines looking at tainted data with string analysis to find vulnerabilities (Gupta, Sharma, Gupts & Gupta, 2012; Wassrmann & Su, 2008). They look for untrusted string values and use formal language methods to find untrusted scripts to discover the root cause of any weaknesses (Wassrmann & Su, 2008).

Lastly, organizations need to setup and monitor their Web-application infrastructure for indications of a possible XSS attack. This can involve configuring security event information management systems to obtain data from intrusion prevention systems, WAF and web servers. Failing to do this risks an organization to attacks and possible data breaches to go unnoticed for a long period of time (Steinke, Tundrea & Kelly, 2011). Thus, the longer a compromise goes unnoticed the more severe the outcome of the attack.

Removing Scripting

Mounika B and Chaitanya (2013) support this claim and add that one needs to deactivate any JavaScript being used in users browsers. However, web applications use Javascript for critical functions of the web site. These include for navigation and presentations. A simple method to combating cross-site scripting could be to stop using any scripting languages in Web sites. These include removing support for various scripting languages such as JavaScript, Perl or VBScript can be done on the server side. This can be easily done but at what cost. These scripts provide support to Web applications and would break them causing the Web site not to function (Harris, 2010). A better method would be to keep untrusted data from active content (OWASP, 2013).
This is accomplished when developers include the option for escaping this untrusted data from the rest of the data being stored in an application. Any active user input data is removed from the content being sent back from the Web server by treating it as text. Thus, any XSS content becomes inert and can’t infect a user’s browser. Another method involves turning off HTTP TRACE support on every Web server (OWASP, 2011). An attacker using HTTP TRACE can post malicious code (URL) that collects cookie information on the user from the server. This information can be sent to another server used by the attacker who now can hijack a session. Turning off the HTTP TRACE support stops the server from performing this activity and protects the users cookie information from being sent to a malicious server.

**Escaping**

Input validation is a common technique used to combat XSS. However, it can’t defend against all types of XSS attacks. To complement this technique, escaping is used. Escaping treats any significant special characters recognized by an interpreter as data and not code (Shar & Tan, 2012; Weinberger, Saxena, Akhawe, Finifter, Shin & Song, 2011). Escaping will only work if the proper escaping scheme is used. These include URL, JavaScript, SQL and HTML. They are used in context to the type of interpreter the browser uses for determining the type of tainted data. So an HTML escape scheme will only work with tainted HTML code. Deploying escaping involves replacing special characters used in HTML into their HTML equivalent ones (Shar & Tan, 2012). This is done by using a standard escaping library in the web application and calling an application programming interface to perform the escape. The process performs a string replacement by invoking an API call to the escaping library (Shar & Tan, 2012).

Common HTML special characters that are exchanged include the “&”, “<” and “>” that are used in developing XSS attacks. For “&” it is replaced by “&amp” so that it is read as data and not a code that could be executed. The “<” is converted to “&lt”, the “>” to “&gt”, and “ to “&quot”. Doing this makes them harmless. So to handle an XSS attack code of “&gt;&lt;” would be converted to &gt;&quot;&lt;&amp;gt; &amp;#47 &amp;#112 &amp;gt. However, it has been found that Java applications are least prone to XSS vulnerabilities Scholte, Robertson, Balzarotti and Kirda (2012a) recommended that the output sanitizer should use complex whitelisting policies to ensure that dangerous characters such as these don’t go through as is. Also the sanitizer should automatically detect ‘ ‘ characters and escape these into a benign attribute. Thus output sanitization can backup input validation by catching any malicious code that gets through input validation.

**Input and Output Filtering**

One of the best defenses against XSS is to perform input and output filtering for malicious content (Mounika B & Chaitanya, 2013; Bisht & Venkatakrishnan, 2008). According to Weinberger, Saxena, Akhawe, Finifter, Shin and Song (2011), filtering is the predominant technique for handling XSS attacks. Performing input validation can be used to discover malicious code. This technique not only helps improve correctness in programs but prevents introduction of security vulnerabilities in applications (Mounika B & Chaitanya, 2013; Scholte, Robertson, Balzarotti & Kirda, 2012b). They use explicit validation processes to check inputs for strings and integers. Filtering the user input for characters that can be seen as HTML tags should
be filtered out (Gupta, Sharma, Gupta & Gupta, 2012; Goodrich & Tamassia, 2011). This validation technique should focus on numeric data and checking of addresses that are in the input strings (Garcia-Alfaro & Navarro-Arribas, 2009). These characters include numeric data and this can help reduce the chances of getting an XSS attack. This method can be done in the browser or server. The browser has no guarantee of integrity that it will compute right to prevent XSS vulnerabilities. Input validation has many advantages. Input validators can prevent XSS vulnerabilities. Also it is simple to cover all types of malicious input data (Mounika B & Chaitanya, 2013). Scholte, Robertson, Balzarotti and Kirda (2012b) therefore recommended doing input validation in the server side.

The process for performing filtering involves two parts: identifying the malicious code and correctly sanitizing it (Weinberger, Saxena, Akhawe, Finifter, Shin & Song, 2011). Filtering the data coming out of the application before being sent to the browser involves knowing what code is considered trusted and what is untrusted. This should be what code the developer built into the application. The second part involves sanitization of the data before it is sent back to the browser. This can be as simple as using escaping techniques to replace the malicious code or scripts into harmless data. Any sanitization must be accurate and complete to prevent an infection.

There are several sanitization options for handling improper inputs. One method uses blacklists to search for improper characters so that they can be replaced or removed (Rao, Tejaswini & Preethi, 2012; Weinberger, Saxena, Akhawe, Finifter, Shin & Song, 2011). This requires the system to know what are bad characters so that they can be dealt with. This technique can be difficult to maintain because today’s attackers can use dynamic sites to conduct their attacks. Another technique restricts inputs to what is also called whitelisting (Rao, Tejaswini & Preethi, 2012; Weinberger, Saxena, Akhawe, Finifter, Shin & Song, 2011). This can be easier to implement because one determines what are good inputs beforehand and blocks all others. However, it can block unlisted inputs that are valid. Scalability can be a problem if done in the browser because each one would have to be updated to ensure they have the latest version (Saha, 2009). Performing this type of filtering in the server can reduce the need to update the browser. The last technique concentrates on searching for escape characters with significant meanings for client-side interpreters (Rao, Tejaswini & Preethi, 2012). This method detects and removes these meanings so as to disable the vulnerability. OWASP provides a valid list of rules that define proper schemes for handling escape characters with HTML.

OWASP (2013) recommends using whitelists for filtering data but doesn’t work well for special characters. This method requires decoding and validating the data before it is inputted. Browsers can use NoScript for establishing whitelists for detecting XSS attacks and sanitizing Web requests (Goodrich & Tamassia, 2011).

Another idea to countering XSS is to use a reverse Web proxy. The proxy is intended to filter traffic from the application server (Garcia-Alfaro & Navarro-Arribas, 2009). The reverse proxy examines the outgoing replies from the server to the user Web browser (Wurzinger, Platzer, Ludl, Kirda & Kruegel, 2009). This way it protects users visiting the Web site from getting malicious code. Wurzinger, Platzer, Ludl, Kirda and Kruegel (2009) created a Secure Web Application Proxy (SWAP) that followed this principal for combating XSS. Their design used a
detection module to find JavaScript in server replies. It contained a sample Web browser to help detect if scripts contain malicious content. Poor filtering policies can negate this and all attackers to overcome the proxy. It can also hurt performance by reducing traffic to and from the server. Another issue is that applications are implemented using different languages and the proxy can’t protect all of these.

Instead of repairing web applications, one can mitigate XSS at the client side browser. Building filters here is easier than at the server side. Client-side XSS filters can mitigate XSS attacks instead of having developers modify the application which can be time consuming (Bates, Barth & Jackson, 2010). This is because the attack code is the same for Web requests and responses from the server (Bates, Barth & Jackson, 2010). This reflected script could be recognized by the browser and blocked. However, one needs to build filters that reduce the number of false negatives. These filters need to examine the semantics of the response in order to reduce the number of false negatives. Performing this after the response is parsed allows the filter to block XSS attacks (Bates, Barth & Jackson, 2010). This is reflected in the following example:

Other filtering techniques include looking at restricting access to malicious URLs on the client-side. This involves using blacklists to identify malicious URLs that redirect users to other Web sites or applications (Garcia-Alfaro & Navarro-Arribas, 2009). However, only basic XSS attacks using the same original URL location can be preventive by this technique. Using a proxy that analyzes the difference between the data being exchanged between the client and server have proved useful. Any malicious code detected is re-encoded by the proxy so as to make it benign (Garcia-Alfaro & Navarro-Arribas, 2009). This technique only detects reflected malicious requests so it can only be used for non-persistent XSS attacks. Thus it is useless against reflected or persistent XSS attacks. Scholte, Robertson, Balzarotti and Kirda (2012b) found that most XSS vulnerabilities can be stopped by using common data types with validation techniques. These common types include integers, Boolean and common strings.

Today attackers may use a variety of technologies such as AJAX, JavaScript, ActiveX, etc. This makes a proxy, using general filtering and analysis-based techniques, useless in detecting malicious code written in these languages (Garcia-Alfaro & Navarro-Arribas, 2009). This also can limit the use of run-time enforcement in browsers. One such method is to audit misuses of operations being performed with the browser. For example, misuse of a JavaScript code that sends an application cookie to unauthorized users would be blocked. Scalability of this approach can be problematic when applying it to complex routines (Garcia-Alfaro & Navarro-Arribas, 2009). A recommended approach is to utilize a policy-based management technique that governs what documents can and cannot do. There is a similar technique called anomaly-based intrusion detection used in clients and servers. In this approach the browser can refuse or allow a script to execute or not. The Browser Enforced Embedded Policies mechanism is one such policy solution by changing the browser so it doesn’t execute malicious code (Gupta, Sharma, Gupta & Gupta, 2012). The policy-based management system does have one drawback, that it can be more restrictive than needed (Garcia-Alfaro & Navarro-Arribas, 2009). This requires one to weigh in on the risk and security of the approach.

The best filtering approach is to use several different techniques. Client-side XSS filters make a good second line of defense for XSS attacks (Bates, Barth & Jackson, 2010). It should be used
with both a server and client side approach. This way different XSS techniques used by attackers can be detected and blocked. This can improve the detection of persistent and non-persistent XSS attacks.

**Web Application Firewall**

Another countermeasure is the use of a Web Application Firewall (WAF). A Web application firewall (WAF) is one countermeasure to use in filtering malicious XSS traffic. It sits between the user and the server to perform checks of traffic patterns. The WAF uses a special description language to implement a security policy for detecting malicious code (Scholte, Balzarotti, Robertson & Kirda, 2012a; Wurzinger, Platzer, Ludl, Kirda & Kruegel, 2009). They are effective in preventing against credentials from being stolen. A WAF differs from a network firewall in that it looks at the application layer of the OSI model to detect XSS codes. The effectiveness of this countermeasure works better with a well designed Web application (Gilad & Herzberg, 2012; Gupta, Sharma, Gupta & Gupta, 2012). Like input filtering, the WAF can be specified in a policy that allows only certain traffic patterns through.

**Patching**

Patching has worked well for software application in protecting against attacks. The same can be done with Web browsers. Therefore, a good countermeasure for persistent and non-persistent XSS is to patch the browsers being used (Gilad & Herzberg, 2012). This will reduce the chances a bug could be exploited by the attacker.

**Education**

Educating developers on this threat can help them become aware of what the XSS can do. This is important since developers may not implement filtering techniques within the software application code. Users should also be aware of the same threat when visiting a Web site. Providing developers a software security guide can help them in recognizing code that would give them help in avoiding vulnerabilities to XSS attacks.

**CONCLUSION**

The threat of XSS attacks are real and can do harm to organizational Web sites. The attacks can be simple to execute and hard to detect because of their flexibility. This paper has examined different techniques that could be used to reduce the risk of this kind of threat. Knowing what XSS is and how they work is just one way of coping with the problem. With XSS being the top vulnerability for Web-based applications one must consider defense to include both the server and client side of the application (Guota, Sharma, Gupta & Gupta, 2012). Thus, the use of multiple countermeasures can provide a layered defense that is better resistant to various types of XSS attack techniques. Current techniques used in defending against XSS have different weaknesses that must be addressed (Rao, Tejaswini & Preethi, 2012). These need to be able to cope with both persistent and non-persistent XSS attacks no matter what type of programming language is used such as PHP, JavaScripts, etc. However, it has been found that Java applications are least prone to XSS vulnerabilities (Scholte, Robertson, Balzarotti & Kirda, 2012b).
There is a need to go beyond the current methods used against XSS and incorporate input validation and sanitization (Rao, Tejaswini & Preethi, 2012). The use of a Web application firewall can provide filtering of web traffic that may be coming from malicious sites. It can also prevent malicious code that may have gotten on a server to be stopped from sending users to malicious sites. The use of input validation checking tools should be utilized to prevent the addition of malicious code being added to the Web application repository. We have shown that both the server and client sides can be done but the client side is far easier. The use of a Web proxy would help with protecting the server side. For the client-side, one should use a policy-based filtering tool to block dynamic XSS attacks. This tool is similar to a host intrusion prevention system that blocks all but normal traffic. Several automated tools were mentioned that could be used for this purpose. Thus, implementing strong security measures can reduce the possibility of information loss and gaining a poor reputation with users.

REFERENCES


