IJCSE International Journal of Computer Sciences and Engineering Open Access
Review Paper Volume-3, Issue-5 E-ISSN: 2347-2693

Reverse Proxy Based XSS filtering

K.S. Wagh, Vishal Jotshi*, Harshal Dalvi, Manish Kamble
Department of Information Technology, University of Pune, Pune

Received: Apr/26/2015 Revised: May/06//2015 Accepted: May/22/2015 Published: May/30/ 2015

ABSTRACT
Due to the increasing amount of Web sites offering features to contribute rich content and the frequent failure of Web developers to properly sanitize user input, cross-site-scripting prevails as the most significant security threat to Web applications. Using cross-site scripting techniques, a malicious user can hijack Web sessions, craft credible phishing sites and using the browser based exploits can have complete access to victim machine. Previous work towards protecting against cross-site scripting attacks suffers from various drawbacks, such as practical infeasibility of deployment due to the need for client-side modifications, inability to reliably detect all injected scripts, and complex, error-prone parameterization. In this paper, we introduce a server-side solution for detecting and preventing cross-site scripting attacks using reverse proxy that intercepts all HTML responses, and allow or deny the request based on filtering techniques using regular expressions and blacklisting techniques.

***Keywords** – HTTP header filtering, Regular expression, Reverse proxy , XSS, XSS firewall,*

I. INTRODUCTION

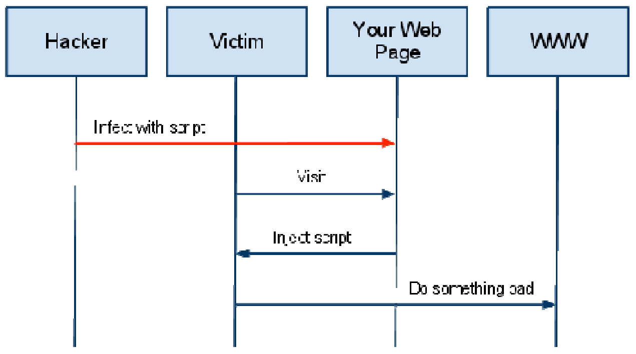
Now-a-days, Cross Site Scripting attacks on websites are rising at a high speed. Different methods are being applied for protection of websites against Cross Site Scripting attacks. Most of the methods are based on individual website protection. But some sites are providing good filters, while the others are lacking protection. So in order to have a common solution for protection against attacks, we are implementing a reverse proxy based XSS filtering firewall which will prevent the XSS attacks on the server side itself.

The problem of XSS arises when a malicious user tries to give malicious input to a web application. This input is an executable code which may get executed on several other clients' browsers. On execution, the code may try to steal legitimate users sessions which may cause unauthorized account access.

II. Cross Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. [1] XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a 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 the browser and used with that site. These scripts can even rewrite the contents of the HTML or dynamic webpage.



A High Level View of a typical XSS Attack

[2]This is a sequential flow of a Cross Site Scripting attack. In this diagram, the hacker first of all finds a vulnerable point in a third party website and tries to inject his own XSS code at the vulnerable point of the

III. Types of Cross Site Scripting

Early on, two primary types of XSS were identified, Stored XSS and Reflected XSS. In 2005, Amit Klein defined a third type of XSS, which he coined DOM Based XSS. These 3 types of XSS are defined as follows:

- **Stored XSS (Persistent or Type I)**

Stored XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser. With the advent of HTML5, and other browser technologies, we can envision the attack payload being permanently stored in the victim's browser, such as an HTML5 database, and never being sent to the server at all.

- **Reflected XSS (Non-Persistent or Type II)**

Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user provided data may never even leave the browser .

- **DOM Based XSS (Type-0)**

As defined by Amit Klein, who published the first article about this issue [3], DOM Based XSS is a form of XSS where the entire tainted data flow from source to sink takes place in the browser, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. For example, the source (where malicious data is read) could be the URL of the page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write)."

IV. Reverse Proxy based filtering

In computer networks, a reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client as though they originated from the proxy server itself. [4] While a forward proxy acts as an intermediary for its associated clients to contact any server, a reverse proxy acts as an intermediary for its associated servers to be contacted by any client.

In our project, We are using reverse proxy based HTTP headers filtering mechanism for detecting the XSS payloads in the incoming requests. This is an efficient method of attack detection.

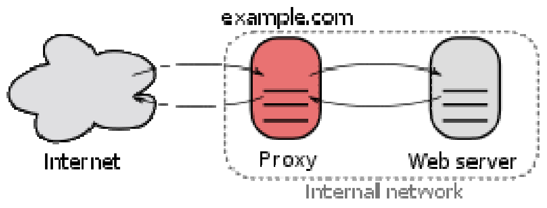


Figure: Reverse Proxy Architecture

[5]Application firewall features can protect against common web-based attacks. Without a reverse proxy, removing malware or initiating takedowns, for example, can become difficult.

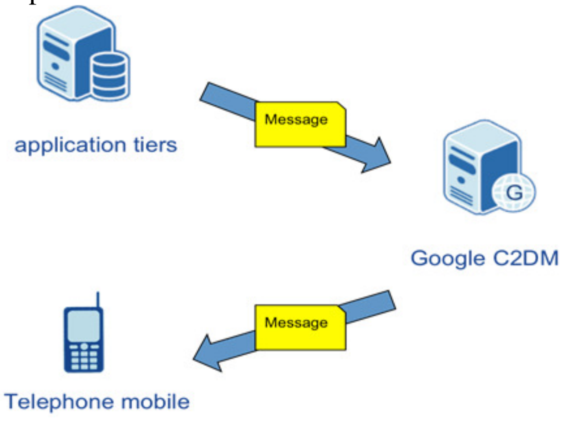
In the case of secure websites, a web server may not perform SSL encryption itself, but instead offloads the task to a reverse proxy that may be equipped with SSL acceleration hardware.

A reverse proxy can distribute the load from incoming requests to several servers, with each server serving its own application area. In the case of reverse proxying in the neighborhood of web servers, the reverse proxy may have to rewrite the URL in each incoming request in order to match the relevant internal location of the requested resource.
Reverse proxies can perform A/B testing and multivariate testing without placing JavaScript tags or code into pages.

V. Android Notification Service

For security auditing purpose, we are using the android notification service to implement the sending of security audit reports from the firewall to the admins android mobile client application. The server admin will be having an android application which will act as

a client and will be connected to the server for fetching the
Audit reports.



The application tier will generate a request if it is having a high priority or the user has demanded the attack report.
This request will be forwarded to an auditing server which will generate the proper format of the report and that report will be deployed to the Administrator.

VI. Cross Site Scripting Payloads

Cross site scripting is having a large number of payloads. Most of them are defined on OWASP as Cheat Sheet which contains a lot of methods and payloads to bypass the XSS filters.

The attack starts mostly when the attacker is successful in closing the parameter string and developing payload as an executable object code.
Here are some methods or events when XSS can occur on a website[6].

- Image XSS using the JavaScript directive
- No quotes and no semicolon
- Case insensitive XSS attack vector
- HTML entities
- Grave accent obfuscation
- Malformed A tags
- Malformed IMG tags
- fromCharCode
- Default SRC tag to get past filters that check SRC domain
- Default SRC tag by leaving it empty
- Default SRC tag by leaving it out entirely
- On error alert
- Decimal HTML character references
- Decimal HTML character references without trailing semicolons
- Hexadecimal HTML character references without trailing semicolons
- Embedded tab
- Embedded Encoded tab
- Embedded newline to break up XSS
- Embedded carriage return to break up XSS
- Null breaks up JavaScript directive
- Spaces and meta chars before the JavaScript in images for XSS
- Non-alpha-non-digit XSS
- Extraneous open brackets
- No closing script tags
- Protocol resolution in script tags
- Half open HTML/JavaScript XSS vector
- Double open angle brackets
- Escaping JavaScript escapes
- End title tag
- INPUT image
- BODY image
- IMG Dynsrc
- IMG lowsrc
- List-style-image
- VBscript in an image
- Livescript (older versions of Netscape only)
- BODY tag
- Event Handlers
- BGSOUND
- & JavaScript includes
- STYLE sheet
- Remote style sheet
- Remote style sheet part 2
- Remote style sheet part 3
- Remote style sheet part 4
- STYLE tags with broken up JavaScript for XSS
- STYLE attribute using a comment to break up expression
- IMG STYLE with expression
- STYLE tag (Older versions of Netscape only)
- STYLE tag using background-image
- STYLE tag using background
- Anonymous HTML with STYLE attribute
- Local htc file
- US-ASCII encoding
- META

  -.1 META using data
  -.2 META with additional URL parameter

- IFRAME

- IFRAME Event based
- FRAME

A basic test to find the vulnerability is to locate the parameter where an attacker can put the payload. Eg:

- **XSS Locator**

Inject this string, and in most cases where a script is vulnerable with no special XSS vector requirements the word "XSS" will pop up. Use this URL encoding calculator to encode the entire string. Tip: if you're in a rush and need to quickly check a page, often times injecting the depreciated "<PLAINTEXT>" tag will be enough to check to see if something is vulnerable to XSS by messing up the output appreciably:

```
';alert(String.fromCharCode(88,83,83))//
';alert(String.fromCharCode(88,83,83))//
";

alert(String.fromCharCode(88,83,83))//";
alert(String.fromCharCode(88,83,83))//--

></SCRIPT>">'><SCRIPT>alert(String.fromC
harCode(88,83,83))</SCRIPT>
```

- **XSS locator 2**

If you don't have much space and know there is no vulnerable JavaScript on the page, this string is a nice compact XSS injection check. View source after injecting it and look for <XSS verses &lt;XSS to see if it is vulnerable:

```
'';!--"<XSS>=&{()}
```

- **No Filter Evasion**

This is a normal XSS JavaScript injection, and most likely to get caught but I suggest trying it first (the quotes are not required in any modern browser so they are omitted here):

```
<SCRIPT
SRC=http://ha.ckers.org/xss.js></SCRIPT>
```

- **Malformed A tags**

Skip the HREF attribute and get to the meat of the XXS... Submitted by David Cross ~ Verified on Chrome

<a onmouseover="alert(document.cookie)">xxs link</a>

or Chrome loves to replace missing quotes for you... if you ever get stuck just leave them off and Chrome will put them in the right place and fix your missing quotes on a URL or script.

<a onmouseover=alert(document.cookie)>xxs link</a>

- **fromCharCode**

If no quotes of any kind are allowed you can eval() a fromCharCode in JavaScript to create any XSS vector you need:

```
<IMG
SRC=javascript:alert(String.fromCharCode
(88,83,83))>
```

- **Embedded tab**

Used to break up the cross site scripting attack:

```
<IMG SRC="jav	ascript:alert('XSS');">
```

These kind of payloads have higher chances of getting past from firewalls.

VII. XSS PREVENTION RULES

The following rules are intended to prevent all XSS in your application. While these rules do not allow absolute freedom in putting untrusted data into an HTML document, they should cover the vast majority of common use cases. You do not have to allow **all** the rules in your organization. Many organizations may find that **allowing only Rule #1 and Rule #2 are sufficient for their needs**.

Do NOT simply escape the list of example characters provided in the various rules. It is NOT sufficient to escape only that list. Blacklist approaches are quite fragile. The whitelist rules here have been carefully designed to provide protection even against future vulnerabilities introduced by browser changes.

- **RULE #0 - Never Insert Untrusted Data Except in Allowed Locations**

The first rule is to deny all - don't put untrusted data into your HTML document unless it is within one of the slots defined in Rule #1 through Rule #5. The reason for Rule #0 is that there are so many strange contexts within HTML that the list of escaping rules gets very complicated. We can't think of any good reason to put untrusted data in these contexts. This includes "nested contexts" like a URL inside a JavaScript -- the encoding rules for those locations are tricky and dangerous. If you insist on putting untrusted data into nested contexts, please do a lot of cross-browser testing and let us know what you find out.

```
 <script>...NEVER PUT UNTRUSTED DATA HERE...</script>   directly in a script

  <!--...NEVER PUT UNTRUSTED DATA HERE...-->   inside an HTML comment

 <div ...NEVER PUT UNTRUSTED DATA HERE...=test />   in an attribute name

 <NEVER PUT UNTRUSTED DATA HERE... href="/test" />   in a tag name

 <style>...NEVER PUT UNTRUSTED DATA HERE...</style>   directly in CSS
```

Most importantly, never accept actual JavaScript code from an untrusted source and then run it. For example, a parameter named "callback" that contains a JavaScript code snippet. No amount of escaping can fix that.

- **RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content**

Rule #1 is for when you want to put untrusted data directly into the HTML body somewhere. This includes inside normal tags like div, p, b, td, etc. Most web frameworks have a method for HTML escaping for the characters detailed below. However, this is absolutely not sufficient for other HTML contexts. You need to implement the other rules detailed here as well.

```
 <body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>

  <div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>
```

any other normal HTML elements

Escape the following characters with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers. Using hex entities is recommended in the spec. In addition to the 5 characters significant in XML (&, <, >, ", '), the forward slash is included as it helps to end an HTML entity.

& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;
/ --> &#x2F;

- **RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes**

Rule #2 is for putting untrusted data into typical attribute values like width, name, value, etc. This should not be used for complex attributes like href, src, style, or any of the event handlers like onmouseover. It is extremely important that event handler attributes should follow Rule #3 for HTML JavaScript Data Values.

```
  <div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content</div>   inside UNquoted attribute

  <div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content</div>   inside single quoted attribute
```

```
 <div  attr="...ESCAPE  UNTRUSTED  DATA
BEFORE  PUTTING  HERE...">content</div>
inside double quoted attribute
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the &#xHH; format (or a named entity if available) to prevent switching out of the attribute. The reason this rule is so broad is that developers frequently leave attributes unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters, including [space] % * + , - / ; < = > ^ and |.

See the ESAPI reference implementation of HTML entity escaping and unescaping.

```
String                    safe             =
ESAPI.encoder().encodeForHTMLAttribute(
request.getParameter( "input" ) );
```

- **Bonus Rule #1: Use HTTPOnly cookie flag**

Preventing all XSS flaws in an application is hard, as you can see. To help mitigate the impact of an XSS flaw on your site, OWASP also recommends you set the HTTPOnly flag on your session cookie and any custom cookies you have that are not accessed by any Javascript you wrote. This cookie flag is typically on by default in .NET apps, but in other languages you have to set it manually. For more details on the HTTPOnly cookie flag, including what it does, and how to use it, see the OWASP article on HTTPOnly.

- **Bonus Rule #2: Implement Content Security Policy**

There is another good complex solution to mitigate the impact of an XSS flaw called Content Security Policy. It's a browser side mechanism which allows you to create source whitelists for client side resources of your web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources. For example this CSP

Content-Security-Policy: default-src: 'self'; script-src: 'self' static.domain.tld

will instruct web browser to load all resources only from the page's origin and JavaScript source code files additionaly from static.domain.tld. For more details on Content Security Policy, including what it does, and how to use it, see the OWASP article on Content_Security_Policy

VIII. CONCLUSION

After implementing the reverse filtering proxy, the proxy server will accept the connections, filter them and then forward them to the actual server to complete the request.
The filtering module will filter the requests and log the attacks, and will remove the XSS attack payload.
The attack notifications will be sent to the admin`s android mobile.

IX. REFERENCES

[1] "DOM Based Cross Site Scripting or XSS of the Third Kind" (WASC writeup), Amit Klein, July 2005
[2] *Cross Site Scripting Definiton ,Web application Vulnerabilities Wikipedia.*
[3] *http://www.cgisecurity.com/xss-faq* XSS attacks.
[4] Mattison Ward, "*Using A Reverse Proxy To Filter HTTP and HTTPS" , GIAC Security Essentials Certification (GSEC), 2012*
[5] *XSS payloads*, OWASP *Cheat Sheet for xss attacks.*
[6] *XSS prevention Rules,*OWASP rules for XSS.