

DOM-based XSS Filter through Browser Extensions

CS5231 Project Report

Heryandi A0095992J

Lu Fangjian A0040740W

Yang Yuhang A0082237H

Yang Zhaoyu A0040953J

1. **Introduction**
2. **Definition of DOM-based XSS**

The main difference of DOM-based XSS compared to other XSS vulnerabilities such as reflected and stored XSS is that DOM-based XSS attack doesn't have to send any data at all towards the server. The attack happens entirely within the browser as opposed to be dependent on the response from the server.

One prerequisite for DOM-based XSS is that the HTML page uses data from document.location or document.URL or document.referrer without sanitizing it first. This prerequisite, however, may be violated as the attackers get more creative.

1. **Example**

We can easily craft a very simple HTML page with DOM-based XSS vulnerability as follows:

<html><head><title>Test Page</title></head>  
 <body>  
 <script>  
 document.write("Site is at: " + document.location.href + ".");  
 </script>  
 </body>  
</html>

If this page is hosted on:

http://localhost/xss.html

Then we can simply open the page and launch the attack with the following URL:

http://localhost/xss.html#<script>alert('xss')</script>

Since the vulnerability is injected through the fragment part of the URL, it happens entirely inside the browser.

1. **Studies From Other Filters**
2. **XSSAuditor, noXSS, IE8 Filter**

In order to design and implement our own filter, some basic knowledge needs to be obtained from the current implementations. We find several implementations, such as noXSS, IE8 Filter and XSSAuditor. However, noXSS has stopped updating since 2008 and there is no source code available; IE8 Filter never published any useful information. Only XSSAuditor is still updating and it’s built into the Chrome browser. Finally, we decided to focus on XSSAuditor.

XSSAuditor is a filter which is highly integrated into Chrome. TheXSSAuditor is in the pipline of displaying a web page in Chrome. After receiving a web page from the server, Chrome starts to parse the web page information, and construct a structure of the web page. When the parsing is finished by the browser, XSSAuditor jumps in and try to find out whether there are some XSS attacks in the webpage or not. Then the last step is Chrome continues with the rendering process. We can see XSSAuditor uses the pipeline of the Chrome browser, unlike other filters such as noXSS which needs a separate parser to parse the page first.

The advantage is that the filter uses the data parsed by the browser itself, so there are no inconsistency between the page seen by the filter and the page seen by the browser. The two parties can always synchronize. The parser of the filter will slow down the rendering process, because the web pages are parsed twice: the first time is by the parser from the filter, the second time is by the parser from the browser itself.

More importantly, we also learn some implementation details from XSSAuditor and other filters:

* 1. <base> element is very dangerous. By injecting it (or altering the href attribute of an existing <base>), an attacker can cause the browser to external scripts from the attacker’s server if the scripts are designated with relative URLs. The attacker only needs to do one attack to change the base URL, and then the user will simply go to attacker’s website. So the filter should block the request which has <base> URL information in it.
  2. Before the filter tries to find the XSS script from the web page, the web page information needs to be decoded. (e.g. replace %41 with A; replace &amp; with &) Otherwise, the attacker may use the encoding mechanism to bypass the filter. For example, an attacker can bypass the IE8 filter by encoding the injected content in the UTF-7 character set, which is not decoded by the IE8 filter regular expressions.
  3. We cannot just simply match the block

<script>…</script>

because the attaker may not need the </script> tag to do the attack. The example is, in the original website, the code looks like:

<?php echo $GET[“q”]; ?><script>/…/…</script>

If the attacker injects code

<script>XSSAttack /

into the page, the page will become:

<script>XSSAttack /…/…</script>

In this case, the code block

<script>…/

can attack already. Instead of attempting to find the entire script in the request, we can try to find some patterns. We can learn some patterns from other filters, such as:

* <script>(.?)</script>
* src[\r\n]=[\r\n]\\\'(.?)\\\'
* document.

After these patterns are found in request, the request needs to be blocked.

* 1. If the string or a substring of a string is in both the request and the response, this request normally contains a reflective XSS attack and the filter needs to block it. For example, in the request, the attacker sends:

http://localhost/page.html?default=<script>alert(document.cookie)</script>

The script part will be also inside the response from the server.

* 1. The filter should apply to the DOM tree created by the parser, because if we apply the filter to the bytes that comprise the response, it may not be as clear as we exam the DOM tree. DOM tree is more structured, and the information in the DOM tree is more accurate. For example, for the web page uses document.write(), examining the bytes may not find the XSS attack. However, examining DOM tree will get it.

1. **Firefox NoScript and Chrome NotScripts**

We have also taken a look at plugins which, by default, block all scripts from executing until they are specifically enabled by the user. With these plugins, users will have to add the Javascript which is allowed to be executed one-by-one to the whitelist. The granularity of each entry in the whitelist is the domain name of the servers hosting the Javascript scripts.

While this may initially sound like a good idea for security, this causes a lot of trouble to users to add the scripts from their favorite websites to the whitelist one-by-one. However, other than that, there are several problems with this approach:

1. A script which is known to users doesn't necessarily mean the script can be safely executed in all cases.
2. A script which is safe to execute today may not be necessarily safe again tomorrow.
3. Because of the granularity of the whitelist entry, if a user wants to enable only one certain script from a domain name, the user will accidentally enable all scripts from the same domain name. Out of the many scripts hosted on the same domain, several scripts may not be safe to execute.
4. **Browser**
5. **Selection**

We have chosen to implement our own plugin on Google Chrome browser. The reason for choosing this browser is because we are able to temporarily disable Google Chrome's anti-XSS protection (XSSAuditor) by using the command:

chrome.exe --args --disable-xss-auditor

This is a very important feature to test our plugin. If the XSSAuditor is enabled, we will not know which of XSSAuditor and our plugin is responsible for blocking the script.

Disabling anti-XSS protection is, as far as we know, not possible in Mozilla Firefox.

1. **Features**

Chrome has the chrome.webRequest API which allows extensions to intercept the web request at various points in its life cycle. We can then analyze the content of request at the interception to eliminate the attack vectors. Our proposed solution is to intercept the request when it is about to be sent, or to intercept the response when it is about to be received. We are still in the progress of finding out the point that we shall intercept.

1. **Implementation**
2. **Idea**

We have chosen to implement our own plugin on Google Chrome browser. The reason for choosing this browser is because we are able to temporarily disable Google Chrome's anti-XSS protection (XSSAuditor) by using the command:

chrome.exe --args --disable-xss-auditor

This is a very important feature to test our plugin. If the XSSAuditor is enabled, we will not know which of XSSAuditor and our plugin is responsible for blocking the script.

Disabling anti-XSS protection is, as far as we know, not possible in Mozilla Firefox.

1. **Logic**

Chrome has the chrome.webRequest API which allows extensions to intercept the web request at various points in its life cycle. We can then analyze the content of request at the interception to eliminate the attack vectors. Our proposed solution is to intercept the request when it is about to be sent, or to intercept the response when it is about to be received. We are still in the progress of finding out the point that we shall intercept.

1. **Result**
2. **Test cases**

We have chosen to implement our own plugin on Google Chrome browser. The reason for choosing this browser is because we are able to temporarily disable Google Chrome's anti-XSS protection (XSSAuditor) by using the command:

chrome.exe --args --disable-xss-auditor

This is a very important feature to test our plugin. If the XSSAuditor is enabled, we will not know which of XSSAuditor and our plugin is responsible for blocking the script.

Disabling anti-XSS protection is, as far as we know, not possible in Mozilla Firefox.

1. **Screenshots**

Chrome has the chrome.webRequest API which allows extensions to intercept the web request at various points in its life cycle. We can then analyze the content of request at the interception to eliminate the attack vectors. Our proposed solution is to intercept the request when it is about to be sent, or to intercept the response when it is about to be received. We are still in the progress of finding out the point that we shall intercept.

1. **Discussion**
2. **What other filters can do**

We have chosen to implement our own plugin on Google Chrome browser. The reason for choosing this browser is because we are able to temporarily disable Google Chrome's anti-XSS protection (XSSAuditor) by using the command:

chrome.exe --args --disable-xss-auditor

This is a very important feature to test our plugin. If the XSSAuditor is enabled, we will not know which of XSSAuditor and our plugin is responsible for blocking the script.

Disabling anti-XSS protection is, as far as we know, not possible in Mozilla Firefox.

1. **Pros and cons**

Chrome has the chrome.webRequest API which allows extensions to intercept the web request at various points in its life cycle. We can then analyze the content of request at the interception to eliminate the attack vectors. Our proposed solution is to intercept the request when it is about to be sent, or to intercept the response when it is about to be received. We are still in the progress of finding out the point that we shall intercept.

1. **Future Work**
2. **aaaa**

We have chosen to implement our own plugin on Google Chrome browser. The reason for choosing this browser is because we are able to temporarily disable Google Chrome's anti-XSS protection (XSSAuditor) by using the command:

chrome.exe --args --disable-xss-auditor

This is a very important feature to test our plugin. If the XSSAuditor is enabled, we will not know which of XSSAuditor and our plugin is responsible for blocking the script.

Disabling anti-XSS protection is, as far as we know, not possible in Mozilla Firefox.

1. **bbbbb**

Chrome has the chrome.webRequest API which allows extensions to intercept the web request at various points in its life cycle. We can then analyze the content of request at the interception to eliminate the attack vectors. Our proposed solution is to intercept the request when it is about to be sent, or to intercept the response when it is about to be received. We are still in the progress of finding out the point that we shall intercept.

1. **Other Findings**
2. **What other filters do**

We have chosen to implement our own plugin on Google Chrome browser. The reason for choosing this browser is because we are able to temporarily disable Google Chrome's anti-XSS protection (XSSAuditor) by using the command:

chrome.exe --args --disable-xss-auditor

This is a very important feature to test our plugin. If the XSSAuditor is enabled, we will not know which of XSSAuditor and our plugin is responsible for blocking the script.

Disabling anti-XSS protection is, as far as we know, not possible in Mozilla Firefox.

1. **Why document.location.href cannot be overridden**

Chrome has the chrome.webRequest API which allows extensions to intercept the web request at various points in its life cycle. We can then analyze the content of request at the interception to eliminate the attack vectors. Our proposed solution is to intercept the request when it is about to be sent, or to intercept the response when it is about to be received. We are still in the progress of finding out the point that we shall intercept.

**Reference**

noXSS <http://www.noxss.org>  
 <https://addons.mozilla.org/en-US/firefox/addon/noxss>

OWASP <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>  
<https://www.owasp.org/index.php/DOM_Based_XSS>  
<https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet>

NoScript <http://noscript.net>