

DOM-based XSS Filter through Browser Extensions

CS5231 Project Report

Heryandi A0095992J

Lu Fangjian A0040740W

Yang Yuhang A0082237H

Yang Zhaoyu A0040953J

1. **Introduction**
2. **Background**

Nowadays, Internet has become essential as people leverage it for business, e-commerce and entertainment. However, web application has been developed with less attention to security such that numerous websites are vulnerable to attack. There are lots of attacks present such as SQL injection, buffer overflow and so forth. In particular, Cross-site Scripting (XSS), which we tackled in our project, is regarded as a very big issue because it is listed as the top web application security problem from OWASP for several years. It is so widespread not only because the websites are vulnerable, but also because visitors who are browsing are unaware about the attacks.

1. **Definition**

DOM-based XSS is a special type of XSS attacks of which its payload does not need to be embedded at the server side. By modifying the DOM environment merely at the client side, the script in victim’s browser would execute unexpectedly, result in the vulnerability.

1. **Feature**

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.

The response from the server doesn’t have to contain any malicious code in order for DOM-based XSS to succeed. The response page only has to access data that is unsafe in order for the attack to succeed. Examples of the unsafe data that can be accessed from a non-malicious page are document.location or document.URL or document.referrer.

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("Default country is " + document.location.href.substring(

document.location.href.indexOf("default=")+8));

</script>

</body>

</html>

If this page is hosted on:

http://localhost/location\_href.html

Normally, the page is accessed via URL such as:

http://localhost/location\_href.html?default=Singapore

However, attacker can easily perform a DOM-based XSS attack by sending the following URL to victim:

http://localhost/location\_href.html?default=<script>alert(1)</script>

Alternatively, if the browser encodes the URL characters, there is always an advanced way like this:

http://localhost/location\_href.html#default=<script>alert(1)</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 being updated since 2008 and only supports very old version of Firefox and furthermore there is no source code available; IE8 Filter never published any useful information. Only XSSAuditor is still being updated and it’s built into the Chrome browser. Therefore, we decided to focus on XSSAuditor.

XSSAuditor is a filter which is highly integrated into Chrome. XSSAuditor 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 additional parser by filters will also slow down the rendering process, because the web pages are parsed twice: the first time is by the parser from filters, 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 attacker 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.
2. **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. **Approach**
5. **Idea**

We have identified several possible sources of DOM-based XSS, namely:

* HTTP referrer
* URL of the page
* Window name

If any of these possible sources are used by the legitimate Javascript of the page without sanitization, then there is risk of DOM-based XSS. For HTTP referrer, the only way to access it is through document.referrer. As for window name, the only way to access it is through window.name. However, there are many possible ways to access the URL (or parts of it) of the page from the Javascript. We list the possible ways below:

* document.URL
* document.documentURI
* document.location
* window.location

This list may look short; however, document.location and window.location have many properties which can access the URL (or parts of it). We list down the properties in the table below. An example is also given to demonstrate which part of URL is returned. This list and example is taken from https://developer.mozilla.org/en-US/docs/DOM/window.location.

Example URL: http://www.example.com:8080/search?q=devmo#test

|  |  |
| --- | --- |
| **Property** | **Example** |
| hash | #test |
| host | www.example.com:8080 |
| hostname | www.example.com |
| href | http://www.example.com:8080/search?q=devmo#test |
| origin | http://www.example.com:8080 |
| pathname | /search |
| port | 8080 |
| protocol | http: |
| search | ?q=devmo |

For all these properties of location, we can separate those that are harmless and harmful as follows:

|  |  |
| --- | --- |
| **Harmless** | **Harmful** |
| host | hash |
| hostname | href |
| origin | pathname |
| port | search |
| protocol |  |

The reason that some of these properties are considered harmless is that if the value of those properties is set to contain malicious script, then the website visited will be totally different from initial website, making the exploit on a website fails because the user will end up in some other website.

For those considered harmful, it is because if the value is set to contain a malicious script, the user will still end up in the same page. The only exception to this is pathname, which can make user visit other page on the same website. We still consider this harmful because if the other page accesses the pathname, the attack can go through. One example is if the URL is set by an attacker as follows: http://www.example.com:8080/<script>alert(1)</script>. This pathname will very likely be not found in any website, however, if the 404 (or other error) page of any website accesses the pathname, then the attack can still work.

Since the main problem is because the page accesses these harmful sources unsanitized, therefore, our idea is to have a filter which sanitizes these information before the page accesses them. In the next section, we will describe how we are able to achieve this.

1. **Implementation**

Our approach is to implement a Google Chrome browser extension, we call this extension DOMXSSFilter. In this extension, we decide to override the getter function of those items we identified previously. The overridden methods encode the values before the data are accessed by the page. By doing this, when the page accesses those harmful properties, it will get the harmless encoded string instead.

In order to override the getter functions, we try to inject a small amount of Javascript into every HTML page opened by the user. Luckily, Google Chrome browser extension provides a facility for us to run some Javascript before the any other script is run. This facility is called Content Script. An additional configuration is needed in the manifest.json in order to use this facility. The additional configuration is as given below.

"content\_scripts": [

{

"matches": ["<all\_urls>"],

"js": ["/inject.js"],

"run\_at": "document\_start"

}

]

The configuration means that for all URLs, run the code inside inject.js first before any other script is run.

Inside inject.js, DOMXSSFilter need to override the getter function of those harmful items. Our approach is to create a Javascript element and append the Javascript element into the original HTML pages by using appendChild:

var s = document.createElement("script");

s.textContent = ‘our code’;

(document.head||document.documentElement).appendChild(s);

Note that we have to write all of our code in a string instead of a separate Javascript file which is imported by the src attribute (which is a more elegant approach) because if we did the latter, the Javascript code imported will be executed asynchronously which will not guarantee that our code will always be executed first.

After our research, we noticed that location.pathname and location.search has been already encoded by Google Chrome, therefore, we can ignore these two items. The rest of the harmful items that need to be taken care of are listed below:

* document.referrer
* window.name
* document.URL
* document.documentURI
* document.location
* document.location.hash
* document.location.href
* window.location
* window.location.hash
* window.location.href

We override the original getter for the above elements one by one. In our new getter, we try to decode the value until no further decode can be done to prevent duplicate encoding and finally encode the value once and return the encoded value. We chose to use encodeURI() to do our encoding. Most of the getters are very similar.

function encodeStringOnce(str)

{

while(str !== decodeURI(str)) {

str = decodeURI(str);

}

return encodeURI(str);

}

var \_\_url = document.URL;

document.\_\_defineGetter\_\_("URL", function() {

console.log("Get url: " + encodeURI(\_\_url));

return encodeStringOnce(\_\_url);

});

However, for some reasons, there are several items which cannot be overridden normally. These are listed below with each of the workaround:

* document.location.href. Although this property contains the whole URL, we have previously analyzed that only hash, pathname and search part may be harmful. Since pathname and search are already encoded by Google Chrome, we only have to deal with the hash. We work around this by overriding document.location to return a location object which has the hash property already encoded instead. This workaround also makes the getter function of document.location.href not necessary to be overridden.
* window.location and window.location.href. The previous workaround to protect href property does not work for window object since window.location is not overridible as well. Therefore, another approach is needed. We decided to forcefully encode the value of window.location.hash so when window.location and window.location.href access it indirectly, the encoded value will be accessed.

Note that unlike other items, window.location.hash does not redirect its user when set with a new value. Therefore, if another Javascript from the same page set the value and later retrieve the value again, it will expect the new value to be returned. Therefore, just for window.location.hash, we need to override the setter as well. This is not the case for document.location.hash since we don’t override its getter; we only override the getter for document.location.

Now, we have the basic implementation of our filter. We decide to investigate on why the getter function of some of these harmful items is not overridable. We present the result of our investigation in the next section.

1. **Additional Improvements**

First, let’s take a look at a Javascript object. Each Javascript object has properties. Each property consists of a name and a value. The name can be any string while the value can be any Javascript value. Each property has property attributes. These property attributes specify whether the associated property can be written, enumerated and configured. These property attributes are respectively called *writable*, *enumerable* and *configurable*. The value of these property attributes is simply a Boolean.

We can easily examine the value of these property attributes of a Javascript object by simply using the following line of code on the Javascript console of Google Chrome. We also provide several examples of usage.

Object.getOwnPropertyDescriptor(obj, "attrName");

// Examples:

Object.getOwnPropertyDescriptor(window, "location");

Object.getOwnPropertyDescriptor(document.location, "href");

Object.getOwnPropertyDescriptor(document.location, "hash");

For the purpose of our filter, we are especially interested with the *configurable* property. If the *configurable* attribute of an object attribute is *false*, the getter and setter cannot be changed. If the *configurable* property of an attribute is initially *true*, then it is possible to change it to *false*. However, if a *configurable* property of an attribute is initially *false*, then it is impossible to change it to *true*.

In the table below, we summarize the property attributes of several DOM attributes in Google Chrome 27 which we are interested in overriding:

|  |  |  |  |
| --- | --- | --- | --- |
| **DOM Attribute** | **writable** | **enumerable** | **configurable** |
| document.referrer | True | True | True |
| document.URL | True | True | True |
| document.documentURI | True | True | True |
| document.location | True | True | True |
| document.location.href | True | True | False |
| document.location.hash | True | True | True |
| window.name | True | True | True |
| window.location | True | True | False |
| window.location.href | True | True | False |
| window.location.hash | True | True | True |

Now we can see clearly why we are not able to override the getter function of document.location.href but is able to deal with it by overriding document.location instead. It is because the *configurable* attribute of document.location.href is *false* while the *configurable* attribute for document.location is *true*.

We can also see why we have no choice but to forcefully encode the value of window.location.hash to protect window.location and window.location.href instead of using a gentler way of overriding the getter function only. It is because the *configurable* attribute for both window.location and window.location.href is *false*. Luckily, the *writable* attribute of window.location.hash is *true*, so there is still at least some protection by forcefully encoding the value.

Now, knowing the nature of property attributes and the value of the *configurable* attribute of these properties, we can make our filter even better by setting the *configurable* attribute to *false* after we override the getter functions. This will make attacks that exploit the overridability of these properties fail. The code snippet to do that is as follows:

Object.defineProperty(obj, "attrName", {configurable: false});

// Examples:

Object.defineProperty(document, "URL", {configurable: false});

Object.defineProperty(window, "name", {configurable: false});

Object.defineProperty(document.location, "hash", {configurable: false});

1. **Tests and Results**
2. **Disable XSSAuditor**

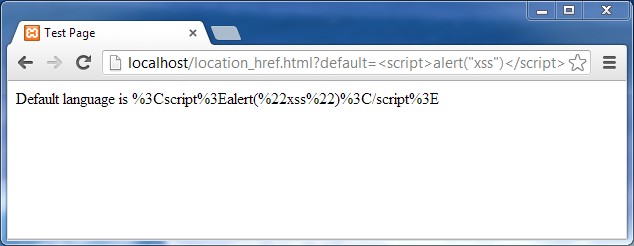
Since we have chosen to implement our own plugin on Google Chrome browser, we can easily disable Google Chrome's anti-XSS protection, XSSAuditor, by using the following command:

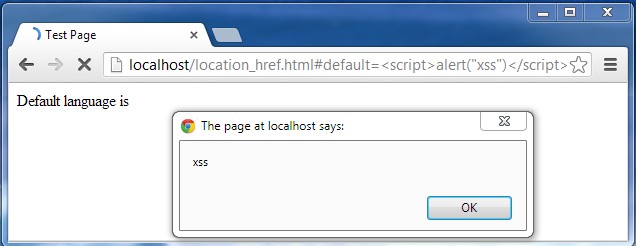
chrome.exe --args --disable-xss-auditor

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.

1. **Testing Technique**

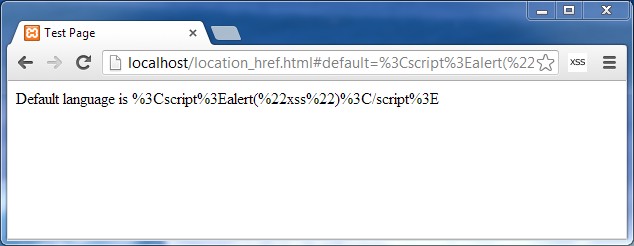
Even with XSSAuditor disabled, the Google Chrome browser still encodes the URL, makes the basic technique of DOM-based XSS attack (using question mark) impossible. However, the advanced technique (using hash) will still work. As such, we will be focusing on the advanced technique during our testing.





1. **Test Result**

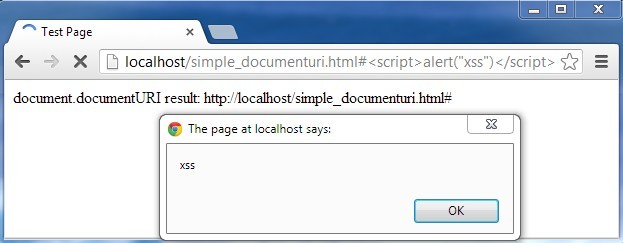
With our extension installed and enabled, we enter the same URL used in the advanced technique from the previous section. As we can see from the result below, the URL becomes encoded, and there is no Javascript pop-up, which means that the attack is successfully blocked.

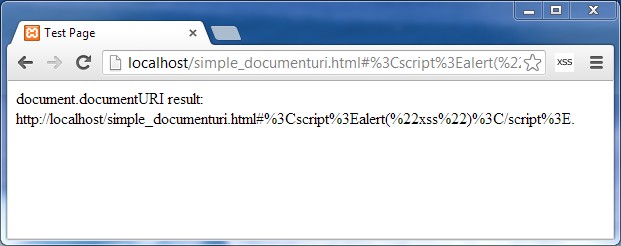


1. **Capability**

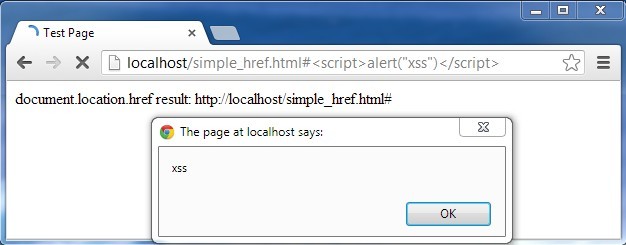
Our extension, however, is not limited to block the attacks only to document.location.href. The attacks that may succeed on other DOM objects, and the capability of our solution to filter them, are illustrated in pairs below. Note that when our extension is installed, it is indicated by the small icon “XSS” on the right of the address bar.

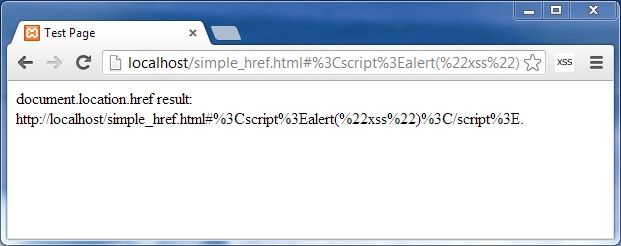
* 1. **simple\_documenturi.html** which exploitsdocument.documentURI



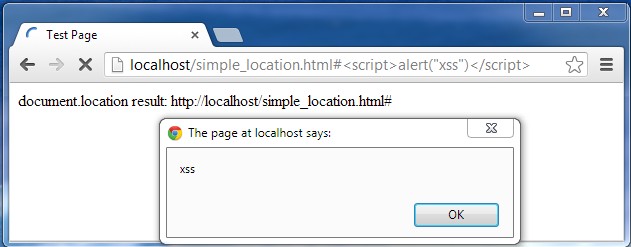


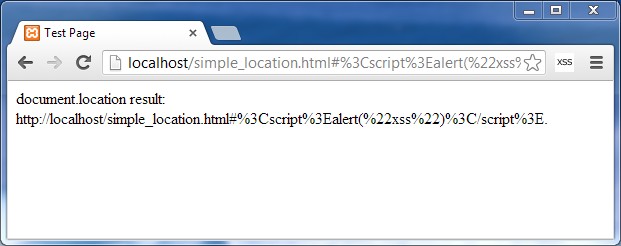
* 1. **simple\_href.html** which exploitsdocument.location.href



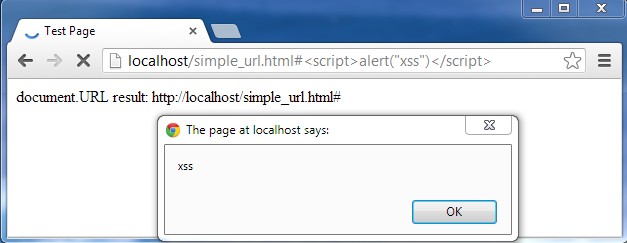


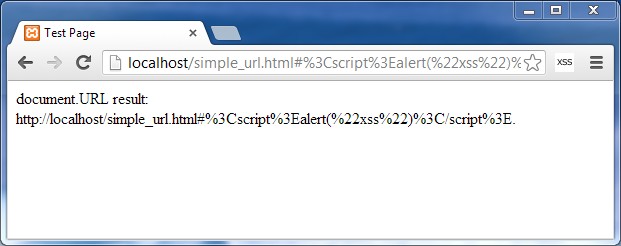
* 1. **simple\_location.html** which exploits document.location



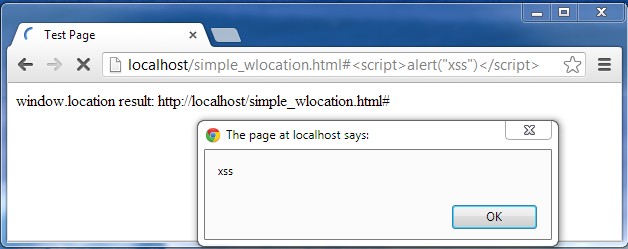


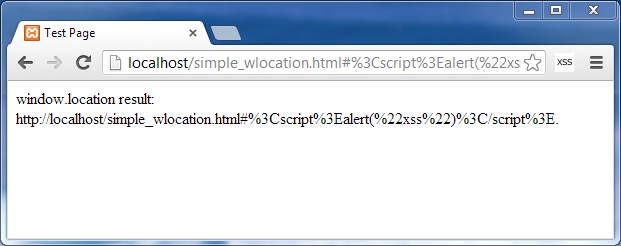
* 1. **simple\_url.html** which exploits document.URL





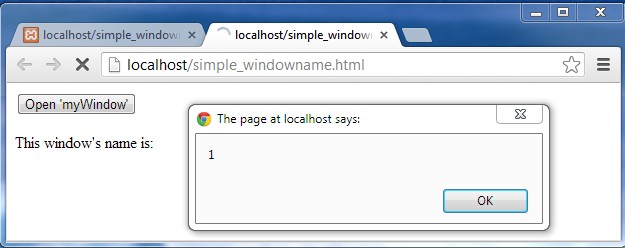
* 1. **simple\_wlocation.html** which exploits window.location

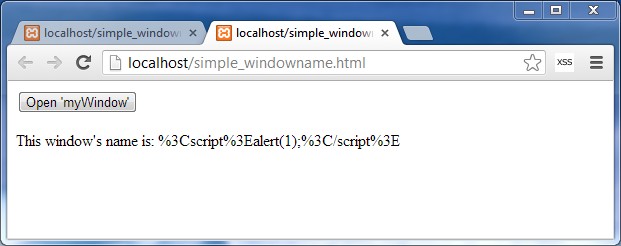




1. **Outstanding feature**

We assess our extension’s capability by comparing it with the default XSS filter of Google Chrome, XSSAuditor. Surprisingly, XSSAuditor does not filter the attacks at a particular DOM object named window.name, which our extension is able to filter it successfully. The two scenarios are illustrated in the following screenshots. The first one shows the result of clicking on ‘Open myWindow’ button in **simple\_windowname.html** when XSSAuditor is used, whereas the second one is when our extension is used. For this particular DOM object, our extension may have outperformed XSSAuditor.





**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>  
<https://www.owasp.org/index.php/Top_10_2013-Top_10>

NoScript <http://noscript.net>