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

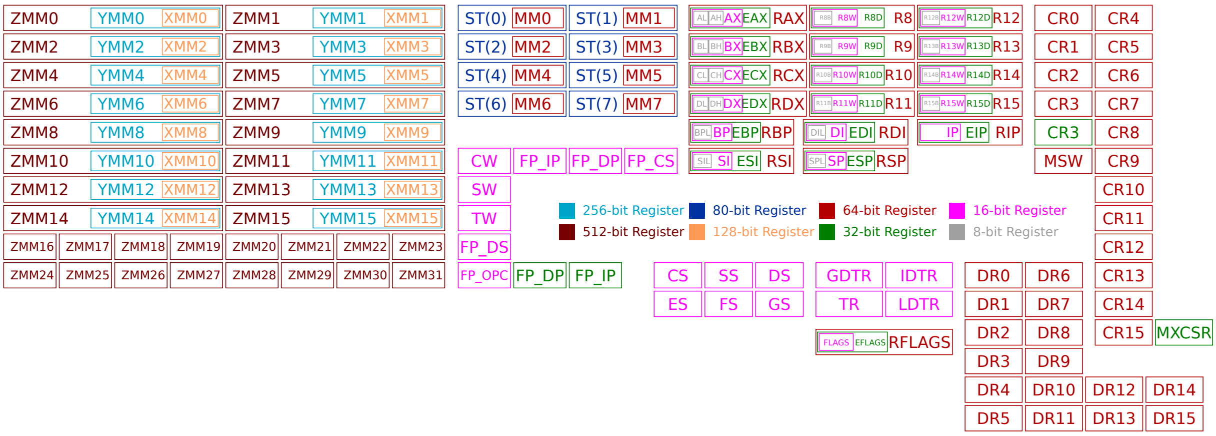
ZMM0 YMM0 XMM0 ZMM1 YMM1 XMM1 ST(0) MM0 ST(1) MM1 AL AH AX EAX RAX R8B R8W R8D R8 R12B R12W R12D R12 CR0 CR4
ZMM2 YMM2 XMM2 ZMM3 YMM3 XMM3 ST(2) MM2 ST(3) MM3 BL BH BX EBX RBX R9B R9W R9D R9 R13B R13W R13D R13 CR1 CR5
ZMM4 YMM4 XMM4 ZMM5 YMM5 XMM5 ST(4) MM4 ST(5) MM5 CL CH CX ECX RCX R10B R10W R10D R10 R14B R14W R14D R14 CR2 CR6
ZMM6 YMM6 XMM6 ZMM7 YMM7 XMM7 ST(6) MM6 ST(7) MM7 DL DH DX EDX RDX R11B R11W R11D R11 R15B R15W R15D R15 CR3 CR7
ZMM8 YMM8 XMM8 ZMM9 YMM9 XMM9 BPL BP EBP RBP DIL DI EDI RDI IP EIP RIP CR3 CR8
ZMM10 YMM10 XMM10 ZMM11 YMM11 XMM11 CW FP_IP FP_DP FP_CS SIL SI ESI RSI SPL SP ESP RSP MSW CR9
ZMM12 YMM12 XMM12 ZMM13 YMM13 XMM13 SW CR10
ZMM14 YMM14 XMM14 ZMM15 YMM15 XMM15 TW CR11
ZMM16 ZMM17 ZMM18 ZMM19 ZMM20 ZMM21 ZMM22 ZMM23 FP_DS CR12
ZMM24 ZMM25 ZMM26 ZMM27 ZMM28 ZMM29 ZMM30 ZMM31 FP_OPC FP_DP FP_IP CS SS DS GDTR IDTR DR0 DR6 CR13
ES FS GS TR LDTR DR1 DR7 CR14
FLAGS EFLAGS RFLAGS DR2 DR8 CR15 MXCSR
DR3 DR9
DR4 DR10 DR12 DR14
DR5 DR11 DR13 DR15
256-bit Register
80-bit Register
64-bit Register
16-bit Register
512-bit Register
128-bit Register
32-bit Register
8-bit Register

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

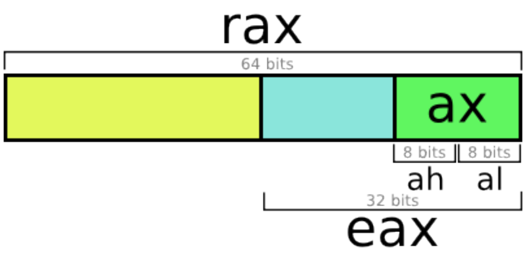

Figure 2: Addressing specific parts of a register including 64 bit extension

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

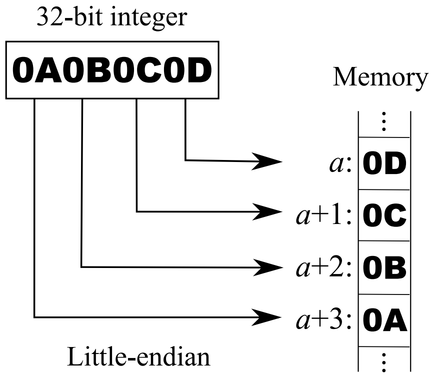

Figure 3: Placement of bytes in memory in little-endian

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. [2, 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

4 Buffer Overflow

5 Shell Code

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