Reverse Engineering

U.S. ARMY CYBER SCHOOL

DEFEND ATTACK EXPLOIT

Version Date: 24 SEP 2018

Student Guide Printable Format

Skills and Objectives

Table of Contents

Understand x86_64 Assembly

Reverse engineering is the fundamental skill underpinning all exploit development and bug hunting. This makes it the fundamental task used to attack and defend any and all systems.

When you think of reverse engineering, do not solely associate it with breaking C programs. Reverse engineering encompasses not only software analysis, but also systems analysis, protocol analysis, and hardware analysis.

Examples:

If you want to remotely control a Honda Civic manufactured after 2016, you need to find a remote exploit. This means that you need to reverse engineer multiple systems within the car.

The cellular data hotspot used for in-vehicle WiFi is run directly in the kernel that controls the entire vehicle. Considering that this is a networked device running in the kernel, it is a good candidate for intial access to completely control the car.

As you continue reverse engineering the car, you will begin to understand how `Controller Area Network (CAN)` works. It operates like a multicast protocol in that it sends all transmissions everywhere in the system. Rogue CAN devices can exploit this and interupt/intercept/change communication on the wire. In fact, US DHS has recently reported on rogue CAN devices: https://www.us-cert.gov/ics/alerts/ics-alert-19-211-01

Considerations:

If you perform reverse engineering, exploit development, and bug hunting before a threat actor does, you are able to implement and embed mitigations directly into the system, rather than adding them on as an after thought. In the above example, a sensible systems engineer, with Security in mind, would not run the cellular hotspot in the kernel. In fact, they would keep it as far away as possible. Prevention of a rogue CAN device may require that CAN wires are not physically accessible, except under maintenance conditions that are not trivial to fake. These conditions would ideally take more time than would be realistically operable as an adversary. CAN may also be protected by authenticating CAN devices.

Why we teach binary reverse engineering:
Reverse engineering compiled C programs is cheap to train and relatable to active missions. Although it is not the end-all-be-all, it follows the same process as all reverse engineering activities. The skills and procedures are directly translatable to all RE mission sets, even though the underlying technologies may change (hardware, protocols, other programming languages, etc)

CPU DIE Explantion and Intro

Utilize labeled cpu die to explain what each part is and what is contained in each component. Each core contains all the transistors that are used to store values for the registers you are about to discuss. Each core also contains an l1 and l2 cache. L3 caches are shared amongst the cores. Caches are the fastest place to store data during operations. L1 is the closest, and therefore fastest to access, but is also the smallest. L2 is slightly larger and slower than l1. L3 shared caches are the largest and slowest to access.

Memory is slower than caches, but substantially larger. The memory controller should be labeled on the diagram.
Refer back to this diagram as necessary. Some students require multiple explanations as to "where" the registers are located and how they are doing their thing.

Programs in Memory

Stack - A group of memory locations; a reserved part of memory used by programmers

When a program is executed, it is loaded into memory. It grows from high to low memory, that is, from user space towards kernel space.
It is broken up into several sections: data, code, stack, and heap. The data and code section contain the referenced data of a program and the code to be executed. The stack and heap are temporary places for temporary storage of things.

The stack follows the principle of last-in-first-out. This means that the last object placed ontop of the stack is the first thing to be removed.
At a high level, the stack can be thought of as a stack of blocks. To add a block to this stack of blocks, you place one on top. To remove something from the stack of blocks, you take the block on top. In this stack of blocks analogy, each block is the same size and the blocks are stack frames. Stack frames each contain data, and are the same size depending on the architecture of the CPU. For a 32 bit x86 processor, the stack frames are 32 bits. For a 64 bit x86 processor, the stack frames are 64 bits.

Understand data sizes

When looking at a disassembled program, you may encounter different sized data references. Below are the most common data sizes that may be seen.

`Bit` - smallest unit in computing. A single 0 or 1
`Nibble` - 4 bits/half an octet
`Byte` - 8 binary bits
`Word` - 16 bits
`DWord` - 32 bits
`QWord` - 64 bits

Understand x86_64 registers

General Register - A multipurpose register that can be used by either programmer or user to store data or a memory location address
There are 16 general purpose 64 bit registers. These registers can be broken down into smaller sections. Take %rbx for example. To call its entire 64 bits, you would use %rbx. However, to call its

lower 32 bits, you would call %ebx. It can then be broken down further into the lower 16 and 8 bits as bx and bl, respectively.
x86 was originally a 32 bit architecture. It originally had eight 32 bit general purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI.

64-bit register	lower 32 bits	lower 16 bits	lower 8 bits	Description
rax	eax	ax	al	
rbx	ebx	bx	bl	
rcx	ecx	cx	cl	
rdx	edx	dx	dl	
rsi	esi	si	sl	
rdi	edi	di	dil	
rbp	ebp	bp	bpl	base pointer; start of stack
rsp	esp	sp	spl	stack pointer;current spot on stack
r8	r8d	r8w	r8b	
r9	r9d	r9w	r9b	
r10	r10d	r10w	r10b	
r11	r11d	r11w	r11b	
r12	r12d	r12w	r12b	
r13	r13d	r13w	r13b	
r14	r14d	r14w	r14b	
r15	r15d	r15w	r15b	

- `%rax` is the first return register. When a function exits and returns a value, or values, the first place it fills is the %rax register.

Question

What would be the first return register of a 32 bit x86 processor?
%eax because it is the lower 32 bits of the %rax register.

- `%rbp` is the base pointer that keeps track of the base of the stack. You will see arguments passed to functions as something like `ivar_8 [%rbp-0x8]`. This is the offset of the memory address that contains argument data passed to the function.

Question

What is the base pointer register of a 32 bit x86 processor?
%ebx because it is the lower 32 bits of the %rbp register.

- `%rsp` is the stack pointer that points to the top of the stack. This moves based on values being pushed to, and popped off, the stack. Keep in mind the 32 bit equivalant %esp.
- All registers can technically be written to as general purpose registers, but you will not see anything outside of standard convention from an industry-standard disassembler.
- The call saved (**non-volatile**) registers are `%rbx`, `%rsp`, `%rbp`, `%r12-%r15`. All other registers are **volatile**. These are registers that will maintain their values even when a function is called or exits.

There is one `instruction pointer register` that points to the memory offset of the next instruction in the code segment:

64-bit	lower 32 bits	lower 16 bits	Descrition
RIP	EIP	IP	Instruction Pointer; holds address for next instruction to be executed

Control Register - A processor register that changes or controls the behavior of a CPU. Control Registers do common tasks like interrupt control, switching the addressing mode, paging control, and co-processor control.
* There are 16 64-bit control registers: `%CR0-%CR15`. Only `%CR0` can be written to or read from. These are generally used in things like Kernel development.

Flags Register - Contains the current state of the processor. FLAGS is 16 bits wide, EFLAGS is 32 bits and RFLAGS is 64 bits wide. The wider flag registers are compatible with the smaller registers.
* There is one 64-bit flags register: `%RFLAGS`.
Flags are set via specific bits in the register. The most important to take note of below are the Carry Flag, Zero Flag, and Sign Flag as these are used by common instructions like JE.
Its `bits` are labeled as:

Bit	Label	Flag Name	Description
0	CF	Carry Flag	Set by arithmetic instructions that generate either a carry or borrow.
1	1	Reserved	
2	PF	Parity Flag	Set by most CPU instructions if the least significant if the destination operand contain an even number of 1's
3	0	Reserved	

Bit	Label	Flag Name	Description
4	AF	Auxilary Carry Flag	Set when a CPU instruction generates a carry to or borrow from the low-order 4 bits of an operand. This flag is used for binary coded decimal (BCD) arithmetic
5	0	Reserved	
6	ZF	Zero Flag	Set by most instructions if the result of an operation is binary zero.
7	SF	Sign Flag	Most operations set this bit the same as the most significant bit of the result. 0=positive, 1=negative
8	TF	Trap Flag	When set by a program, the processor generates a single-step interrupt after each instruction
9	IF	Interrupt Enable Flag	When set, external interrupts are recognized by the processor on the INTR pin. When set, interrupts are recognized and acted on as they are received.
10	DF	Direction Flag	Used for string processing. When set to 1, string operations process down from high addresses to low addresses. If cleared, string ops process up from low to high addresses. Set and cleared using STD and CLD instructions

Bit	Label	Flag Name	Description
11	OF	Overflow Flag	Set by most arithmetic instructions indicating the result was to large to fit in the destination.
12/13	IOPL	I/O Privilege Level	Used in protected mode to generate 4 levels of secuirty: Level 0- Private OS functions; most privileged Level 1- OS Services Level 2- Device Drivers Level 3- Application Programs; least privileged
14	NT	Nested Task	Used in protected mode. When set, it indicates that one system task has invoked another via a CALL instruction rather than JMP
15	0	Reserved	
16	RF	Resume Flag	Used to debug registers R6 and R7. Enables you to turn off certain exceptions while debugging code.
17	VM	Virtual-8086 Mode	Permits 80386 to behave like a high speed 8086.
18	AC	Alignment Check	
19	VIF	Virtual Interrupt Flag	
20	VIP	Virtual Interrupt Pending	
21	ID	ID Flag	
22-63	0	Reserved	

NOTE Certain instructions like `CMP` set these bits, and other instructions read them to determine certain actions like `JE`. The `zero flag` is particularly important.

There are also sixteen `128 bit SSE registers` used for floating point and double operations.

They are `%xmm0` - `%xmm15`

- SSE=Streaming SIMD Extension
- SIMD=Single Instruction, Multiple Data: Typical applications are digital signal processing and graphics processing.

Registers Diagram
Utilise provided registers diagram to visually explain how certain registers can be broken down into different components.

Understand standard x86_64 Intel instructions

`MOV`: move source to destination

```
mov r15,#n  <- r15 = #n
mov rax,m   <- move contents of 64bit address to %rax
mov m,rax   <- move contents of %rax to 64 bit memory address
```

`PUSH`: push source onto stack

```
push r15    <-push r15 onto stack
```

`POP`: Pop top of stack to destination

```
pop r8      <-move value on top of stack to r8
```

`INC`: Increment source by 1

```
inc r8      <-increment value in r8 by 1
```

DEC: Decrement source by 1

```
dec r8      <-decrement value in r8 by 1
```

`ADD`: Add source to destination

```
add r13,#n  <-add #n to %r13, store result in %r13
```

`SUB`: Subtract source from destination

```
sub r13,#n  <-subtract #n from %r13, store result in %r13
```

`CMP`: Compare 2 values by subtracting them and setting the %RFLAGS register. ZeroFlag set means they are the same.

```
cmp r8, r9  <- compare value of r8 to r9. Set flags as appropriate.
```

`JMP`: Jump to specified location

```
jmp MEM1    <-jump to memory label MEM1
```

`JLE`: Jump if less than or equal

```
jle MEM1    <-jump to memory label MEM1 if less than or equal
```

`JE`: Jump if equal

```
je MEM1     <-jump to memory label MEM1 if equal
```

Understand the x86_64 stack

IMPORTANT Explained at the begining of lecture with the diagrams

DEMO: x64 ASM flow

```
main:
    mov rax, 16     //16 moved into rax
    push rax        //push value of rax (16) onto stack. RSP is pushed up 8 bytes (64
bits)
    jmp mem2        //jmp to mem2 memory location

mem1:
    mov rax, 0      //move 0 (error free) exit code to rax
    ret             //return out of code

mem2:
    pop r8          //pop value on the stack (16) into r8. RSP falls 8 bytes
    cmp rax, r8     //compare rax register value (16) to r8 register value (16)
    je mem1         //jump if comparison has zero bit set to mem1
```

```
main:
    mov rcx, 25     //store the value 25 in rcx register
    mov rbx, 62     //store the value 62 in rbx register
    jmp mem1        //jumps to mem1 location

mem1:
    sub rbx, 40     //subtract 40 from rbx
    mov rsi, rbx    //copy rbx value to rsi
    cmp rcx, rsi    //compare the values in rcx and rsi
    jmple mem2      //jumps to mem2 location if value is less than or equal

mem2:
    mov rax, 0      //store 0 in rax
    ret             //return out of code
```

=== Follow C Source Code

```
Unresolved directive in lesson-6-reverse_sg.adoc - include::example$csource1.c[]
```

```
Unresolved directive in lesson-6-reverse_sg.adoc - include::example$csource2.c[]
```

Python Programming*

DEMO: Python Programming

```
Unresolved directive in lesson-6-reverse_sg.adoc - include::example$pythondemo.py[]
```

```
Unresolved directive in lesson-6-reverse_sg.adoc -
include::example$pythonDemoFunctions.py[]
```

```
Unresolved directive in lesson-6-reverse_sg.adoc - include::example$demo.py[]
```

Binary Disassembly with Standard Industry Tools

IMPORTANT This document was made available to students to follow in the CTFd.

Recieving or discover object to be analyzed

This is the phase where reverse engineering and analysis begins. You have recieved the object (software, hardware, signals) to be reverse engineered. This could happen in any number of ways.
Examples of acquisition:

- An object, or software for our case, is sent to your team by another group.
- Your team may have have found malicious software in traffic.
- A piece of enemy hardware may have been captured.
- You have captured traffic of some kind with unique protocols that need to be analyzed and reverse engineered.
- Think of any other methods of acquisition of software or hardware.

Determmine reason for analysis
This is where you determine, or re-evaluate, your intent behind analyzing the object.
Are we analyzing with the intent of determining if it’s malicious?
Are we intent upon retooling potential malware for our own purposes?
Are we attempting to create mitigations and signatures?
Are we attempting to find vulnerabilities in an effort to create exploits for later use?
Are we attempting to find vulnerabilities in an effort to create mitigations/patches to harden our networks?

These questions will help an analyst determine what they need to do to accomplish their goals and will ultimately change what they document, focus on, and look for.

Static Analysis
IMPORTANT: We are focused on software analysis here. The steps for hardware or protocols would be different given the conditions.

Initial static analysis of a binary gives an analyst, or team of analysts, several clues as to what the binary is designed to do and how it really works.

This is where demos begin:

1. Determine file type - Is it an executable? What environment is it designed to run in? (OS,cpu architecture, etc)
   This can be accomplished in many ways:
   1. Open CFF explorer and load in the executable (demo.exe or other). The top tab on the left menu allows for viewing the file type/architecture and compiler.
   2. Open executable file in a raw text editor and view header. MZ is the header for a portable executable which is generally associated with Windows executables. ELF is the header for an Executable and Linkable Format file which is generally referred to as a Linux executable.
   3. Run the "file" command on the binary on a Linux box.
2. Determine if file is packed/compressed (UPX)
   1. With CFF explorer opened with the executable file loaded in, scroll go to the UPX/packing tab located on the left menu. If it does not have the option to unpack, the file is not packed with UPX. Packing is outside of the scope of this class, but it is a compression technique that

    allows for moderate obfuscation.

3. Find plain text ascii and unicode strings
    1. Utitlize CFF EXplorer strings on the document to find ascii strings. CFF Explorer may not be configured to find all ascii strings or any unicode strings.
    2. IDA free and Ghidra may be configured to find unicode strings and ascii strings that CFF Explorer may not.
4. View sections of the executable to find potential obfuscation.
    1. Using CFF Explorer, open the "section headers" tab on the left menu. Look for anything out of the ordinary. You would expect to see .txt, .rdata, .data, .rsrc, and .reloc. Anything else would point to obfuscation and may even be name with the type of obfuscation (i.e. AESeal)
5. View imports/exports to get a hint of functionality/calls (is it importing a library that can open a socket, etc?) 5. Look for encrypted sections of the binary
    1. Utilize CFF Explorer and view contents of the "imports" and "exports" tabs on the left menu.
        - The imports tab can give hints as to what functionality the executable is utilizing. If you see a library being called that deals with networking or sockets, then you can dig into that and view what functions from it are being referenced. This would indicate that the binary is utilizing sockets either for IPC or some kind of networked operation which may mean that it is calling back to the malicious author or many other things.
        - The exports tab will contain things that the binary is making available to other things on the system. This is common for things like malicious DLLs.
6. View resources to find embedded objects
    1. Utilize CFF Explorer to view the "Resource Editor" tab on the left menu. This will allow analysts to dig into the many resources of the binary. The resources can be extracted through CFF Explorer if they seem interesting.

A good reference for CFF Explorer is this: https://medium.com/@tstillz17/basic-static-analysis-part-1-9c24497790b6

Behavioral Analysis
Behavioral analysis of software (or anything) is generally considered the most basic form of analysis. Simply put, the thing being reverse engineered is interacted with in a variety of ways and allowed to behave as normal. Reactions are recorded with various utilities and analyzed.
Because of the ease and speed of this form of analysis, it is generally completed by less experienced analysts as a "triage phase".

Behavioral Analysis is the fastest way to gain insight into how a binary works.

1. Take a snapshot of the analysis environment - Important! Taking a snapshot on an OpenStack VM takes a substantial amount of time.
2. Take a snapshot of critical configurations of the analysis environment. (Things like the registry, important directories, etc)
3. Launch realtime analysis tools (Things like procmon and fakenet)
4. Execute and interact with the object/binary while taking note of observations.

5. Stop the binary and view how it affected critical configurations (registry, files, etc) by comparing to previous snapshots
6. Analyze results of realtime analysis tools (did fakenet catch network calls, did procmon show it writing to a file, etc)

Because the class moves so fast, and our environment is not setup to be a proper reverse engineering shop, we will limit behavioral analysis to simple interactions with the binary. This is all that will be required to complete the activities in this class.

To demo behavoral analysis:
1. Open the binary that you have been analyzing with CCF Explorer.
2. Take note of what happens. Save the strings you see on the terminal to a text document. Ensure students take note that the program has "printed" these strings to the terminal.
3. Enter some gibberish and record what happens. Save strings to your notes. These "artifacts" will be used later in analysis.
4. Take note that the program prints out a failure statement and then "Sleeps" for 5 seconds. This failure statement and sleep behavior can be used later in analysis.
5. Repeat step 1-4, but incorporate sysinternal tools such as procmon. Set the filter to for only the process that you're analyzing. Inform students that you are looking for "writing" to registry keys, creating/reading files, and potential process forking.

Dynamic Analysis
Dynamic analysis for software anlaysis usually refers to debugging. During this type of analysis, a debugger is attached to a process, or vice-versa, and keeps tracks of all of the memory, registers, and other important components of a running piece of software.
Debuggers do not fully unpack or disassemble a piece of software and relies solely on how a binary is being interacted with.

Dynamic analysis is similar to behavioral, except the analyst is attaching the process to a debugger.
Debuggers are very effective when dealing with encryption or obfuscation. This is because the binary will potentially deobfuscate itself and reveal its obfuscation operations or encryption key.

1. Execute binary in a debugger
2. Step through binary, setting breakpoints as appropriate
3. Continuously rerun the binary and edit its parameters through the debugger, as you learn more about how it works
4. Document all observations and modifications

Disassembly
An analyst may be required to disassemble a binary to learn more about how it runs.

1. Disassemble binary in Ghidra
    1. Open Ghidra, start a new project, and import the binary that is being analyzed (demo.exe or other). Once imported, the splash screen for the binary should include information about its

    file type, cpu architecture, and several other things. This should match what you have found so far (x86, 32 bit, portable execuatble)

    2. Run the code analyzer by clicking on the dragon icon. If your binary does not get loaded into the code analyzer, import it by going to "File"→"Import" or "Open".
    3. When asked, accept the default options and "analyze" the binary. This automatic analysis finds lots of things that may otherwise not be seen by the disassembler.

2. Use notes to find artifacts within the disassembly
    1. Ask students what artifacts you have already found to help direct your analysis of the disassembled binary. They should respond with some of strings printed to the terminal. Start with one of those strings.
    2. Go to View → Search For → Strings. Edit the options to your liking (default works fine). All the found strings will be displayed. The bottom of this window contains a search bar. Search for your chosen string artifact. Clicking on the string artifact will take you to the data section of the disassembly. You will need to find the "xref" to the string and click on that. This "xref" will take you to the code portion of the disassembly that it is calling.
3. Find a good spot to work from within the binary. Then quickly browse from the top to the bottom of the disassembly to view the overall flow of the disassembly
    1. You have already established where to work from. If you chose a failure message or the initial prompt string, it will put you in the "main" function where you will see the initial printf statement ("Enter key: ") and failure message. Right above the failure, and now visible success, message will be the conditions that have to be met to be successful.
4. Work your way from the bottom to the top - if there are two outcomes choose the one you want to end at, then work your way up from there to determine what needs to happen for the program to flow to the desired outcome. Rename variables and functions as appropriate when quickly scanning top to bottom of the disassembly.
    1. Once you have found the success message, work upwards. There will be a condition that has be met. This will be a comparison between the return code of the most recent function and some value. This value will be in hex but can be converted by right-clicking on it in the disassembly and converting to unsigned int.
    2. Take note of the value you need to be successful then go into the most recent function to determine under what conditions it will return the value you need for the success message. If you are using demo.exe, you will need to look several functions deep, but at the end of the day, the binary is taking the user input, converting it to integer with atoi(), and subtracting that value from 123. If the difference between userinput and 123 is zero, you will return a success statement.
    3. Test out what you have just found by running the binary, entering an acceptable key (123 for demo.exe), and showing the students a success message.

A Professional Shop

A professional reverse engineering shop will follow something similar to this exact process. The only difference is that there will be different individuals or teams assigned to different tasks.

A general overview of a professional shop workflow:

1. Initial analysis will be performed by "triage analysts". This phase will involve the first several steps of analysis after the shop recieves the binary and their tasking. Shop leadership will have determined the end goal for the analysis. Triage analysts will then take the binary and the goal and perform initial static and behavioral analysis. They will document everything they find and submit their reports/evidence/findings back to the team. Their report may include things like general functionality of the program, a determination of if the binary is malicious, potential IP addresses that the binary calls back to, and potential additional functionality they may not have been able to confirm.
2. As appropriate, mid-level analysts will take the findings of the triage analysts, and expand upon them. They will use all the evidence and artifacts found by triage analysts to speed up their dynamic analysis and disassembly. Mid-level analysts will generally verify all findings from the triage analysts, and perform more detailed analysis. This step in a professional shop confirms or refutes the findings of the triage analysts, document additional functionalities, and produces some signatures to detect the binary behaviors on the wire or on a host machine.
3. When requested or necessary, "in-depth analysts" or "senior analysts" will perfrom extremely detailed analysis of the binary. They will use and confirm the previous analysts' findings, and greatly expand the understanding of the binary. This phase will generally produce advanced signatures, extremely detailed report on the exact functionality of the binary, and new tools/scripts to be used by analysts for future analysis. In-depth analysts will also regularly work with developers to retool malware, patch vulnerabilities in software, and create exploits for operational use.

How malware and executables use encryption
Payload encryption: A portion within the binary is encrypted then injected into another process or dropped as an executable payload onto the system.

Section encryption: Commonly referred to as a crtyper. These encrypt data or code sections of an executable.

Binary packing: The binary is compressed. Can generally be unpacked with open source tools.

Regardless, the binary must go through this cycle: decrypt/unpack, drop/run payload, cleanup, cease execution. When analyzing the binary, this should be easy to find/see either in disassembly or a debugger.