LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

LINUX CNO USER MODE DEVELOPMENT

USER MODE DEVELOPMENT MODULE

ACTP
ADVANCED CYBER TRAINING PROGRAM

LINUX CNO USER MODE

SERIES

- After the Linux CNO user mode Series, the student will be prepared to engage in usermode CNO development tasks in Linux
- Assessments:
  - Daily Quizzes
  - 14 Labs
  - Final Assessment
    - Minimum score of 80%

Labs

- 14 labs
  - Do not count towards pass/failure of course
  - All solutions will be posted
- We are here to help

- Lab Tips:
  - Remember your tools: `valgrind`, `gdb`, `objdump`, `readelf`, etc.
  - Spend time understanding how each system works before you try writing code to manipulate it

Quizzes and Final Assessment

- You are strongly encouraged to build, compile, and execute any code that might help answer questions
- Multiple choice questions…
- Quizzes do not count towards passing/failure of the course
  - But are recorded
- All conversations will be taken outside during assessments

Series Agenda

I. CNO User Mode Development Overview
II. CNO Assembly Programming
III. Mobilized Code
IV. Injection
V. Hooking
VI. Hardening

ACTP
ADVANCED CYBER TRAINING PROGRAM

Learning Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

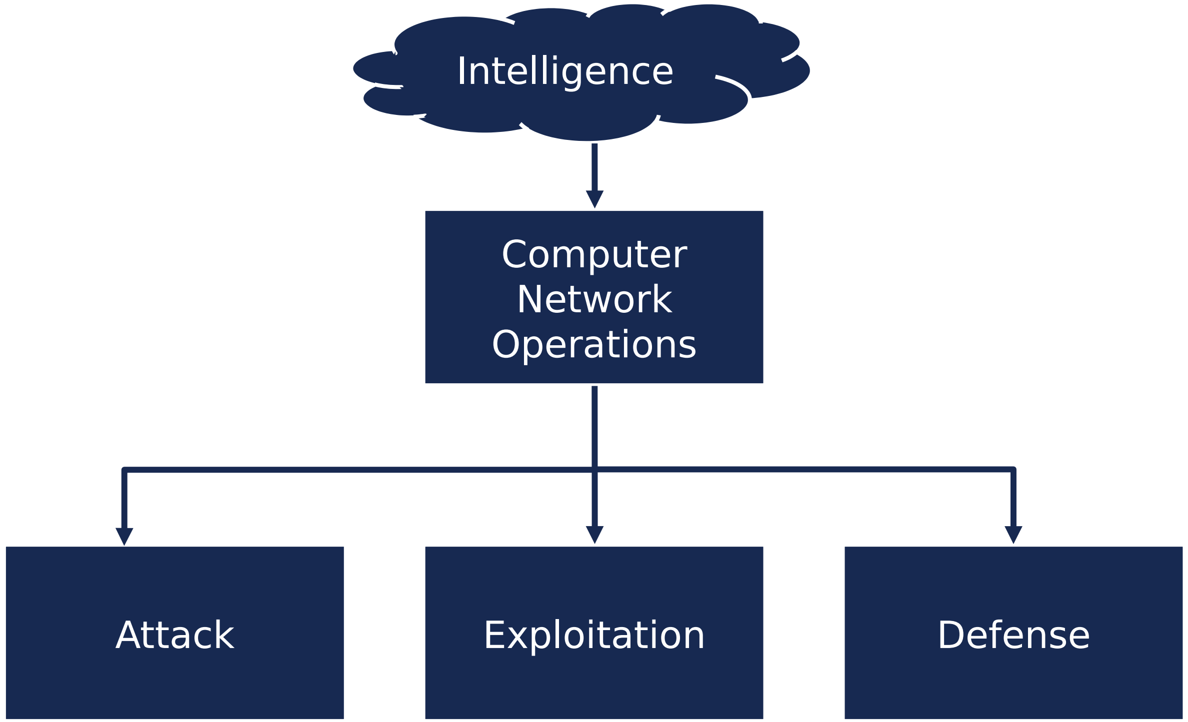
- Understand the scope and nature of CNO development in userspace
- Understand assembly concepts and how they apply to CNO development
- Understand what mobilized code is and how to write it and use native tools to generate it automatically
- Understand injection and be able to perform injection tasks for programs on disk and in memory
- Understand hooking and perform hooking tasks through linker manipulation, callback registration and code patching
- Understand and engage in hardening practices including hiding, obfuscation, red herring, and others

What Now?

ACTP ADVANCED CYBER TRAINING PROGRAM

- In the previous two classes we explored programming in Linux and studied the internals of Linux from the perspective of userspace applications
- In this class we will use the knowledge and skills gained from those previous classes to develop payloads that manipulate processes and hide such activities from system services
- In the next class, we will explore methods of gaining execution to launch such payloads, through vulnerability research and exploitation
- After that we'll start exploring all these topics in kernelspace

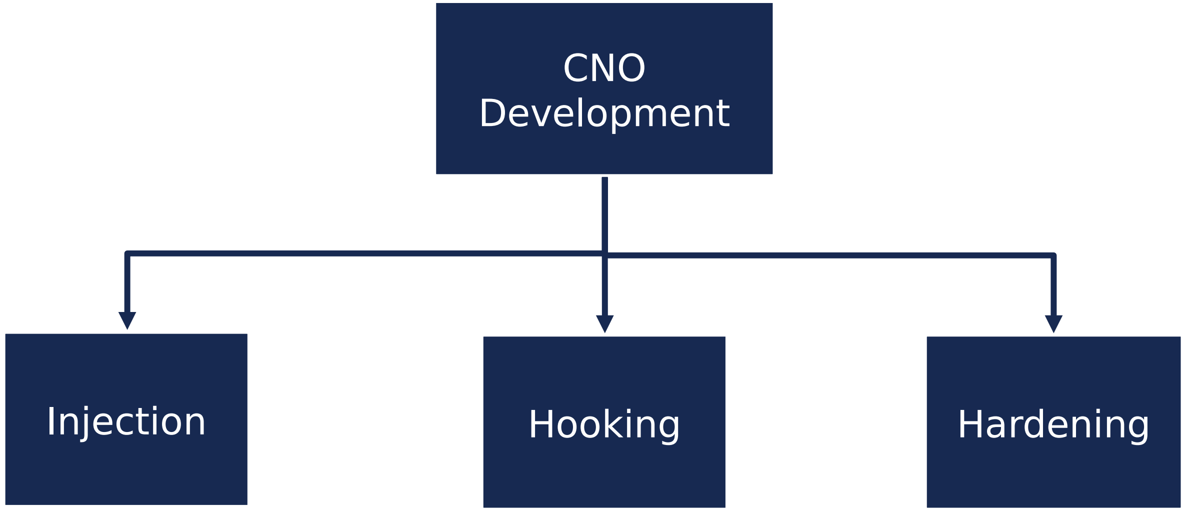
What is CNO?


Intelligence
Computer Network Operations
Attack
Exploitation
Defense

What is CNO Development?

- Computer Network Operations (CNO) are focused on technologies to defend, attack, and exploit computer networks
- A CNO developer is responsible for creating and maintaining these technologies
- CNO development is an ongoing endeavor that requires ingenuity and creativity, and successful developers draw on a deep understanding of internal system architecture and behavior
- We will be exploring some core CNO topics and using a variety of sample methodologies to deliver hands-on experience with each one

What is CNO Development?



Preliminary Skills

- Before we begin we need to introduce some concepts and tools that will be useful for CNO development on Linux:
  - GNU Assembler (with AT&T syntax)
  - GCC inline assembly & extended assembly
  - Position independent code
  - Mobilized code
  - GNU linker scripts

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

GNU Assembler

Hit the GAS

GCC and GAS

- GCC uses the GNU Assembler (GAS) as its default backend for assembly
- GAS is an implementation of the Unix Assembler, as
- The original as only supported AT&T syntax (no Intel syntax support)
  - Up until version 2.10, GAS only supported AT&T syntax too
  - As a result, the Linux kernel and many other systems use AT&T syntax
- There are some differences between AT&T and Intel syntax:
  - Source and destination operands are flipped
  - Registers are prefixed with % and constants are prefixed with $
  - Mnemonics have a suffix to indicate the size of their operands
  - Effective addresses use parenthesis(base, index, scale)

AT&T vs Intel Assembly

- Examples:

AT&T	Intel
movl %eax, %ebx	mov ebx, eax
movl $56, %esi	mov esi, 56
movl %ecx, 8(%edx, %ebx, 4)	mov [edx + ebx*4 + 8], ecx
movb %ah, (%ebx)	mov [ebx], ah
addq %r9, %rax	add rax, r9

- Note: the size suffixes (b, w, l, q, etc.) in AT&T syntax are not necessary if the size can be inferred from the destination operand

The GNU Assembler

- **GAS**
  - as -o program.o program.s
  - Default AT&T Syntax
  - Can generate ELF objects
  - Can be linked with ld
  - Supports Macros / Preprocessor
  - Works with lots of Architectures
  - Part of GCC toolchain
  - Assembles GCC’s inline assembly
- **NASM**
  - nasm -f elf -o program.o program.asm
  - Default NASM Syntax
  - Can generate ELF objects
  - Can be linked with ld
  - Supports Macros / Preprocessor
  - Works with x86, 16 bit, 32 bit, 64 bit

The GNU Assembler

GAS	NASM
`# Comment (for x86)` `/* Multiline Comment */`	`; Comment`
`.byte 0x0`	`db 0x0`
`.asciz "Zero terminated String"`	`db "Zero terminated String", 0`

`mystr: .ascii "Some Str"`
`mystr_end:`
`.set MYSTR_SZ, mystr_end - mystr`
	`mystr: db 'Some Str'`
	`MYSTR_SZ equ $ - mystr`

`.macro mymacro arg1 arg2`
`    xor \arg1, \arg2`
`.endm`
	`%beginmacro mymacro 2`
	`    xor %1, %2`
	`%endmacro`

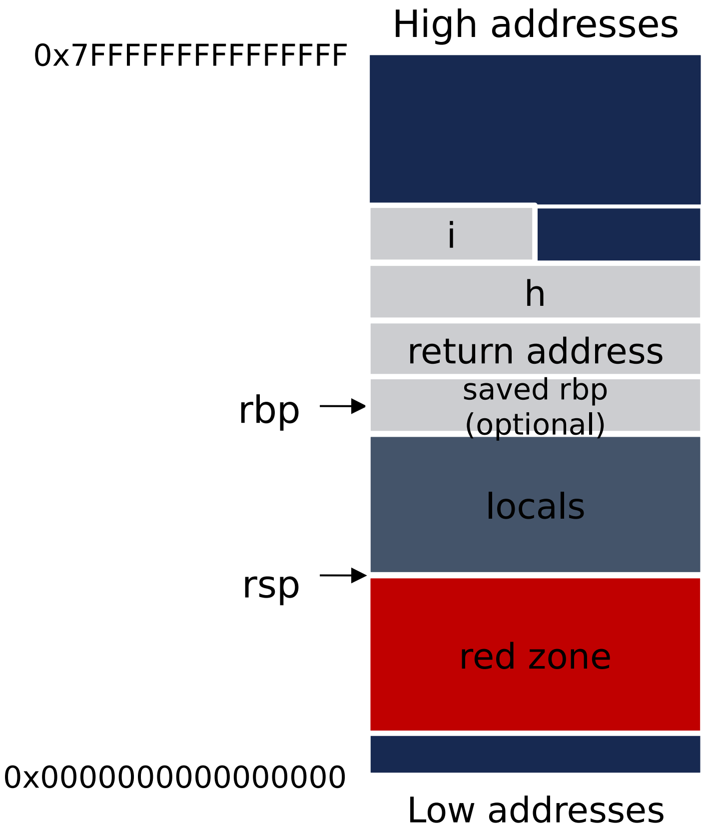
`.global someLabel`	`GLOBAL someLabel`

Linux x86_64 Calling Convention

- Called the "System V AMD64 ABI"
- First Six integer or pointer arguments
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - "`%r10` is used for passing a function's static chain pointer" (For nested functions)
  - Volatile registers (caller-save): `%rax, %rcx, %rdx, %r8, %r9, %r10, %r11`
- `%xmm0 – %xmm7` used for floating point arguments.
- Rest of argument's on the stack (pushed in reversed order)
- Stack aligned on 0x10 boundary
- Return in
  - `%rax` (64 bit integral)
  - `%rdx` and %rax (128 bit integral)
  - `%xmm0` (double)
  - `%xmm0` and %xmm1 (quadruple)
- Linux will not touch a 128 byte "red-zone" space just beneath (toward 0) the stack pointer.
  - Leaf-node functions can use this area without fear of getting clobbered by signal handlers and such
  - Microsoft x64 uses required 32 bytes of "shadow space" right before a call
    - Not the same as red-zone
- Functions with variadic argument lists (...) requires number of floating point arguments in al register
  - Cannot pass in floats to many glibc vararg functions (`printf`), they must be converted to doubles first (`cvtss2sd`)

Linux x86_64 Calling Convention

```
long func(long a, int b, char* c, float* d, double e, long f, long g, long h, int i);
...
func(a,b,c,d,e,f,g,h,i);
```



INTEGER REG	VAL
%rdi	a
%rsi	b
%rdx	c
%rcx	d
%r8	f
%r9	g

XMM REG	VAL
%xmm0	e

Linux x86_64 syscall Calling Convention

- Number of the `syscall` in `%rax`
- Arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Volatile registers (caller-save): `%rcx`, `%r11`
- Result in `%rax`
  - A value between the range of -4095 and -1 indicates an error
    - error return is a `–errno`
- All system calls only have up to 6 arguments, no arguments are passed on the stack

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 1

Assembly with GAS

Tasks

- Create a .S file that will:
  - Implement the two function declarations found in `student.h`
  - Invoke a system call using `syscall`
  - Call the glibc function `printf`
- Tips:
  - File descriptors 0, 1, and 2 are usually standard in/out/err, respectively
  - Create string literals to help
    - `.data`
    - `MYSTRING: .asciz "mystrcontents"`
  - In gdb use `stepi` (`si`) and `nexti` (`ni`) as opposed to `step` (`s`) and `next` (`n`)
    - steps instructions as opposed to source lines

ACTP
ADVANCED CYBER TRAINING PROGRAM

GCC Inline Assembly

ManTech

GCC Inline Assembly

- GCC allows you to include assembly directly inside your C files
- The `asm` GNU extension is used to provide raw assembly instructions
- Example:

```
__asm__ ("movl %eax, %ebx\n\t"
      "movl $56, %esi\n\t"
      "movl %ecx, (%edx, %ebx, $4)\n\t"
      "movb %ah, (%ebx)\n\t");
```

- Since the `asm` keyword is a GNU extension, use `__asm__` instead for standard-compliant code such as ANSI

GCC Inline Assembly *(continued)*

- GCC provides extended inline assembly to provide more control and options over inline assembly
  - Read and write to C variables by name
  - Perform jumps from assembly code to C labels
- Some limitations apply to extended inline assembly
  - functions declared with the naked attribute must only use basic inline assembly
  - Any assembly outside of a function needs to be basic inline assembly

Extended Inline Assembly

ADVANCED CYBER TRAINING PROGRAM

- Typical extended inline assembly format:

```
asm asm-qualifiers ( AssemblerTemplate
     : OutputOperands
     [ : InputOperands
     [ : Clobbers ] ])
```

- Goto form:

```
asm asm-qualifiers ( AssemblerTemplate
     :
     : InputOperands
     : Clobbers
     : GotoLabels)
```

Extended Inline Assembly

ADVANCED CYBER TRAINING PROGRAM

Can list multiple instructions on multiple lines (separated with \n\t).

```
__asm__ ("mov %[input0], %[output0]\n\t"
         "mov %[input1], %[output1]\n\t"
         : [output0] "=&r" (output0), [output1] "=r" (output1)
         : [input0] "r" (input0), [input1] "r" (input1)
         :);
```

Can list multiple instructions on a single line (separated with \n\t or ;)
Can also have blank/empty operand lines.

```
__asm__ ("mov %[input0], %[output0];" mov %[input1], %[output1]"
         :
         :
         :);
```

Extended Inline Assembly *(continued)*

- asm-qualifiers
  - `volatile` – tell GCC that your assembly has side effects
  - `inline` – GCC makes resulting assembly as small as possible
  - `goto` – tell GCC that your `asm` statement may perform a jump
- AssemblerTemplate
  - String literal containing assembly, possibly mixed with tokens
- OutputOperands
  - Comma-separated list of C variables modified by the assembly
  - Can be empty
- InputOperands
  - Comma-separated list of C expressions read by the assembly
  - Can be empty

Extended Inline Assembly *(continued)*

ACTP
ADVANCED CYBER TRAINING PROGRAM

- Clobbers
  - Comma-separated list of registers and memory modified by the assembly
  - Can be empty
- GotoLabels
  - Comma-separated list of all C labels to which assembly can jump
  - Only used in the goto form for extended inline assembly

Assembler Templates

- Assembler Templates are the actual inline assembly you wish to embed into your C program
- Note that some additional assembly will be generated by the compiler, based on the input/output operand constraints, clobber list, etc.
- Symbolic names can be placed inside your assembly, to bind them to specific C variables later
- Example template: `"movl %%eax, %[myvar]\n\tcli"`
  - Note that we have escaped %eax by prefixing it with another %, because % is also the token for symbolic names (ugh, ugly)
  - Note that multiple assembly statements should be separated with “\n\t” (newline and tab) to match standard assembly syntax

Extended Inline Asm Example

```
uint32_t rotate_right(uint32_t num, uint8_t bits) {

    __asm__ volatile ("mov %[n], %%cl\n\t"
                      "rorl %%cl, %[regA]"
                      : [regA] "+r" (num)
                      : [n] "rm" (bits)
                      : "cc", "%cl"
    );
    return num;
}
```

Assembler template. Here we load some operand (n) into register %cl (note we escape the % with another %). We then rotate another operand (`regA`) right by the number of bits specified by %cl

Asm-qualifier (optional)

Output operands

- When specifying your list of output operands, you can give the compiler commands about how to handle each output
- Each output operand takes the following form
  - [asmSymbolicName] constraints (cvariablename)
- asmSymbolicName - References a symbolic name given to this operand in the assembler template. **Optional**
- constraint – dictates to the compiler where to put this operand (e.g., what register) and how it is modified
- cvariablename - A C lvalue expression to hold the output (like a local variable name)

Extended Inline Asm Example

```
uint32_t rotate_right(uint32_t num, uint8_t bits) {

    __asm__("mov %[n], %%cl\n\t"
        "rorl %%cl, %[regA]"
        : [regA] "+r" (num)
        : [n] "rm" (bits)
        : "cc", "%cl"
    );
    return num;
}
```

Output operands.
Here a single output operand is specified. We tell the compiler to map the output `regA` to a any register GCC chooses, mark it was read and written (+), and bind its value to the C variable `num`

Input Operands

- When specifying your list of input operands, you can give the compiler commands about how to handle each input
- Each input operand takes the following form
  - [asmSymbolicName] constraints (cvariablename)

asmSymbolicName - References a symbolic name given to this operand in the assembler template. **Optional**

- constraint – dictates to the compiler where to put the operand before the inline assembly is executed (it will generate code to do this for you)
- cvariablename - A C expression that currently holds the value to be used as the input (like a local variable name)

Extended Inline Asm Example

```
uint32_t rotate_right(uint32_t num, uint8_t bits) {

    __asm__("mov %[n], %%cl\n\t"
        "rorl %%cl, %[regA]"
        : [regA] "+r" (num)
        : [n] "rm" (bits)
        : "cc", "%cl"
    );
    return num;
}
```

Input operands.
Here a single input operand is specified. We tell the compiler to map the input [n] to a register OR memory (whichever it pleases), and bind its value to the C variable `bits`

Some Common Constraints

- Output operand constraints must begin with either ‘=‘ (meaning overwriting) or ‘+’ (reading and writing) – will NOT be both ‘=+’

CONSTRAINT	MEANING
r	Use any register for this operand
m	Use memory for this operand
rm	Use a register or memory, whichever is more efficient
0,1,2,3,etc.	Make this operand the same as the Nth operand
Some x86-specific constraints	
a	Use the a register (a, ax, eax, rax)
b	Use the b register (b, bx, ebx, rbx)
c	Use the c register (c, cx, ecx, rcx)
d	Use the d register (d, dx, edx, rdx)
S	Use the si register (si, esi, rsi)
U	Use a call-clobbered integer register

Clobbers

- The compiler is made aware of direct changes to register and memory values through the use of the input and output operand lists, but the inline assembly may modify more than this
- To notify the compiler about (and allow it to properly protect) other modified values, you add them to the comma-separated clobber list
- Each item can be a register name, “cc” (for flags register), and/or simply “memory”
- Example: “cc”, “%rax”, “b”, “%r10”

Extended Inline Asm Example

```
uint32_t rotate_right(uint32_t num, uint8_t bits) {

    __asm__("mov %[n], %%cl\n\t"
        "rorl %%cl, %[regA]"
        : [regA] "+r" (num)
        : [n] "rm" (bits)
        : "cc", "%cl"
    );
    return num;
}
```

Clobbers.
Here we tell the compiler that, in addition to the input and outputs, the **assembler template** clobbers the flags (cc) and %cl registers

Goto Labels

- Goto labels are used when an assembler template can result in a jump to some other portion of code
- Is a comma-separated list of all the symbol names to which the assembly may jump

Extended Inline Asm Example

```
uint32_t rotate_right(uint32_t num, uint8_t bits) {

    __asm__("mov %[n], %%cl\n\t"
        "rorl %%cl, %[regA]"
        : [regA] "+r" (num)
        : [n] "rm" (bits)
        : "cc", "%cl"
    );
    return num;
}
```

Assembler template. Here we load some operand (n) into register %cl (note we escape the % with another %). We then rotate another operand (`regA`) right by the number of bits specified by %cl

Clobbers. Here we tell the compiler that, in addition to the input and outputs, the **assembler template** clobbers the flags (cc) and %cl registers

Input operands. Here a single input operand is specified. We tell the compiler to map the input [n] to a register OR memory (whichever it pleases), and bind its value to the C variable `bits`

Output operands. Here a single output operand is specified. We tell the compiler to map the output `regA` to a any register GCC chooses, **mark it was read and written (+)**, and bind its value to the C variable `num`

Further Reading

- There are plenty of other constraints (especially architecture-specific ones) and other features of extended inline assembly
- GNU.org has a great resource for learning more about GCC's extended inline assembly
  - https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Extended-Asm.html

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 2

Extended Inline Assembly

Tasks

- Create a C program that obtains information about the local machine’s CPU
- Create a C function that uses GCC extended inline assembly to issue the `cpuid` instruction to obtain this information
- Read the `cpuid` entry in the intel software developer’s manual to determine the inputs and outputs of this instruction
  - See page Vol 2A 3-191 (PDF page 293)
- You do not need to handle all possible inputs. Just print out human-readable output for EAX = 0 and EAX = 1

Tasks *(continued)*

- Your C program should only need to use one line of explicit assembly (but you can do it with more, if you'd like)
- Save the output of the `cpuid` instruction into four separate C local variables, and use these in `printf` statements to make a human-readable result

Tasks *(continued)*

- When finished, your program print out things similar to the following:

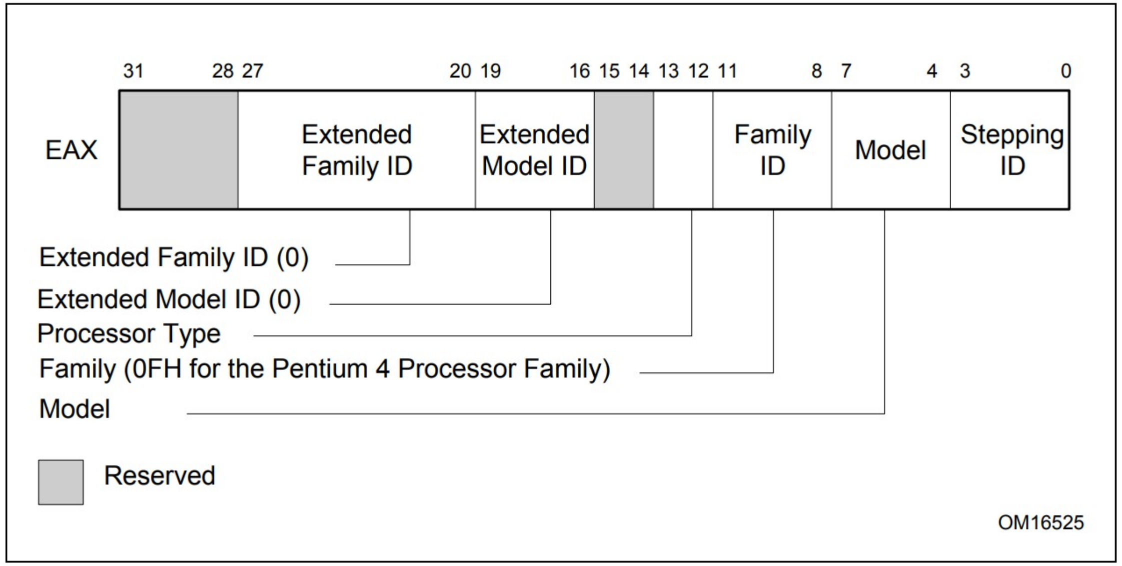
```
$ ./extended_asm 0
Basic CPU information associated with input: 0
GenuineIntel

$ ./extended_asm 1
Basic CPU information associated with input: 1
Family:          6
Model:           14
Extended Model       5
Brand index:       0
Cache line size (bytes): 64
```

Tasks *(continued)*

- Answer the following questions
  - Is it possible to do this using only basic inline assembly?
  - What is another way to do this?
  - What are the pros and cons of each approach?
- Hints: Think carefully about your input operands and output operands before you even write C or assembly. What variables do you want bound to specific input registers? What output registers do you want bound to C variables?

Tasks *(continued)*

- The following chart may be helpful for EAX = 1



LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Mobilized Code

ManTech

Position Independent Code

- What makes code position *dependent?*
  - Absolute addressing vs relative
    - Mobilized code has no relocations that need to be done before code is run
- x86_64 has some useful tools that we don't have in 32 bit
  - `%rip` relative addressing
- Pointers in the data section are always going to need to be relocated
  - Function tables
  - Global pointers
    - Example: `char* mystr = "hello, my name is";`
- Pointers to data section are fine, though
    - Example: `char mystr[40];`

Position Independent Code *(continued)*

Source

```
_start:

  call label           #; compiler defaults to relative
                       #; call has max 32-bit offset
                       #; there is no 64-bit direct call

  lea label, %rax      #; not relative

  lea label(%rip),%rax #; relative version of above

  call *indlabel       #; uses absolute address of indlabel

  call *indlabel(%rip) #; uses offset to indlabel
                       #; but indlabel contains an absolute
                       #; address (so this is not PIC)
label:
  ret

indlabel:
  .8byte label
```

After Linker

```
0000000000401000 <_start>:

401000: e8 1c 00 00 00          callq 401021 <label>


401005: 48 8d 04 25 21 10 40    lea 0x401021,%rax
40100c: 00
40100d: 48 8d 05 0d 00 00 00    lea 0xd(%rip),%rax

401014: ff 14 25 22 10 40 00    callq *0x401022

40101b: ff 15 01 00 00 00       callq *0x1(%rip)


0000000000401021 <label>:
401021: c3                      retq

0000000000401022 <indlabel>:
401022: 21 10                   and %edx,(%rax)
401024: 40 00 00                add %al,(%rax)
401027: 00 00                   add %al,(%rax)
```

Position Independent Code *(continued)*

- Position Independent Code (PIC):
  - Machine code that can run at an arbitrary memory location
    - Does not have absolute internal references
    - Does not contain relocation information
- GCC already has support this for various use cases
  - Shared Objects
  - PIE binaries
- Mobilized Code:
  - PIC with the ability to be injected into and run within arbitrary buffers
  - Why is being able to run at any memory location so important for CNO work?

What is it used for?

- Mobilized Code used for
  - Shellcode
  - Injected stubs
  - Hooks
- Stages other code for:
  - Acting in the other process/thread's context
  - In-process hooks
  - Covertly subverting and/or performing surveillance on processes

Creating Mobilized Code

- Mobilized code can be generated through
  - Handmade assembly
  - GCC's tool chain
    - `-nostdlib`
    - Use of static libraries (`-static`)
    - `-pie -fPIE`
    - `-fPIC`
    - linker scripts

Creating Mobilized Code

- Hand-crafted assembly
  - Traditional “shellcode” is made this way
  - We’ll show some techniques during the labs that will help to do this
  - **Minimal size; non-trivial to develop, change, and maintain**
- Compiler-generated code
  - C/C++ code that is compiled to assembly for us
  - **Easy to develop, maintain, and change; size tends to be larger than hand-coded**
  - GCC option `-fPIC` or `–fPIE`
    - PIE = Position Independent Executable
      - For executables
      - Uses PC-relative (think %rip) relocations
    - PIC = Position Independent Code
      - For libraries
      - Uses a PLT (we’ve seen this)

Mobilized Code – Loader

- A loader is a bit of mobilized code that will load a program/library into a process without having to drop anything to disk
- To load an ELF, iterate through program headers and map every `PT_LOAD` type into memory, and then call the entry address
  - If the program is type `ET_DYN`, then the interpreter needs to be loaded and run too
    - Usually `ld-linux.so.2`
    - This will perform relocations and linking
- Normally we would just call `dlopen`, but `dlopen` requires a path to a file to load
  - We don't have one if we're using mobilized code

Mobilized Code – memfd_create

- One helpful tool that Linux provides is `memfd_create`
  - "... unlike a regular file, it lives in RAM and has a volatile backing storage."
- With `memfd_create` we can get a file that is never mapped to a file system
  - Unless it is swapped out to the swap space
- We can pass `/proc/self/fd/<memfd>` to `dlopen` and load our shared object that way
- This example is a way to create a simple loader without having to parse the ELF and dynamically link it yourself

Mobilized Code – Linker Scripts

- When creating mobilized code, we often want to control what is placed where
- `ld`, the GNU linker, is amazingly customizable
- Every link is controlled by a linker script
  - use `ld --verbose` to view the default linker script
    - This script changes with commands passed to `ld`
      - Try `ld -z now -z relro -shared -fPIC --verbose`
  - We can supply our own linker script using `–T`
- You can specify the input files using command line arguments
  - Example: `ld -o output.bin -T linkscript.lds input_a.o  input_b.o`

Mobilized Code – Linker Scripts

- What can a linker script do?
  - Position sections in file
  - Position sections in virtual memory
  - Discard sections
  - Mark sections as loadable or not
  - Create positional variables that can be used by the code being linked
    - You can create variables for the start and end of a function, used to checksum your code
  - Output to a certain format (binary, ELF, PE, etc.)
  - Allocate regions of memory
- Why do we care?
  - Payloads can rely on executable code being the first thing in the output binary
  - We want to remove sections that don't make sense for our use case
  - We want to pack other payloads into the binary easily and reference them in our code

Simple Example Linker Script

```
OUTPUT_FORMAT(elf64-x86-64)
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : {
    mylib.o(.data)
    myotherlib.o(.rodata)
    myotherlib.o(.custom_sec)
  }
  .bss : { *(.bss) }

  /DISCARD/ : {
    *(.debug*)
    mylib.o(.note*)
  }
}
```

Specifies the binary output format (64-bit ELF)

Indicates that the succeeding sections should start at the given address

Indicates that the .text section will be made up of all the input files' .text sections

Indicates that the .data section will be made up of mylib.o's .data section, myotherlib.o's .rodata section, and myotherlib.o's .custom_sec section

Indicates that the .bss section will be made up of all the input files' .bss sections

Discard all input files' sections whose names start with .debug. Also discard all of the sections of mylib.o whose names start with .note

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 3

Run me once, shame on me

Run me twice, shame on you

Tasks

- This lab provides a example loader to you. Use it to load a .so of your own into a vulnerable application over the network.
- Objectives:
  - Load your .so over a network connection
  - Understand the Loader provided to you
- Bonus:
  - Modify the linker script to link in the .so payload at compile time, instead of in a separate step
  - Have your payload create a reverse shell

Tasks *(continued)*

- The Loader is provided in `ldr.S` and `lds_lib.c`, with a `linkerscript.ld` and `Makefile`
  - `ldr.S` is the entry point
    - It expects that immediately after the compiled buffer is a 4-byte length field
    - The length field is to be immediately followed by a .so of that length
    - `ldr.S` takes these parameters, and calls into `ldBuf` which does the actual loading
  - `ldr_lib.c` contains most of the loading logic
  - `linkerscript.ld` links the loader together into a cohesive binary unit

Tasks *(continued)*

- The vulnerable application is `nasmline`
  - The source of the program is provided to you
  - It is (foolishly) being hosted on a remote port
  - Find and exploit the vulnerable condition in the code in order to invoke the loader and load a .so file of your creation

LINUX CNO Programming

ACTP ADVANCED CYBER TRAINING PROGRAM

Usermode Injection

Libraries, Elves, Maps, and More

What is Code Injection?

- Modifying a process or context to execute something it was not expecting to execute
- It is useful for:
  - Stealing Secrets
    - Encryption Keys, Session Tokens, Passwords
  - Modifying behavior
    - Why implement your own client when you can use theirs?
    - Stealing privileged connections and handles
  - Debugging
    - Many debuggers require code injection to work
      - Breakpoints in x86 place a 0xCC over an existing byte to cause an interrupt. (Hardware breakpoints do not need to do this)

Injection in Linux

- Not as nice as windows
  - No clean API to create a remote thread
- Requires certain permissions to interact with another process
  - Default usually requires process to have same euid and ruid, and have the "dumpable" flag set

"Various parts of the kernel-user-space API (not just ptrace() operations), require so-called "ptrace access mode" checks... based on factors such as the credentials and capabilities of the two processes, whether or not the "target" process is dumpable, and the results of checks performed by any enabled Linux Security Module (LSM)"

-ptrace(2) man page

Injection in Linux *(continued)*

- A few Methods:
  - `LD_PRELOAD`, `LD_LIBRARY_PATH`
    - Replacing dynamically loaded symbols
  - ELF Poisoning
    - Modifying binaries without clobbering code
  - `/proc/<pid>/mem`
    - Reading / Writing to another virtual address space
  - `ptrace`
    - Debugging a shared library into a running process

Injection with LD_PRELOAD

- Dynamic linker/loader (`ld.so` / `ld-linux.so`)
- See `man ld.so`
- `LD_LIBRARY_PATH`
  - Will look for shared libraries in these paths before other places
- `LD_PRELOAD`
  - Will load these libraries first for symbol resolution
  - In secure-execution mode pathnames with slashes ignored
- These are ignored in secure-execution mode
  - Secure-execution mode enabled by
    - euid != uid (effective UID different from real UID)
    - A process with a non-root UID executed with capabilities
    - a Linux Security Module (LSM)

LD_PRELOAD Example

- You can set the `LD_PRELOAD` variable for a single program using the shell:
  - `LD_PRELOAD=./myoverride.so ./myprogram`
- You can set `LD_PRELOAD` for the whole session
  - `export LD_PRELOAD=./myoverride.so`
- You can also set `LD_PRELOAD` globally
  - Whitespace separated list of entries in `/etc/ld.so.preload`

Injection with LD_PRELOAD

Pros
- Non-invasive
- Low overhead

Cons
- Only replace dynamically linked functions
- Doesn’t work* with setuid binaries
- Requires ability to influence the environment
- Doesn’t work on already executing binaries

*This does work if the following conditions are met:
The LD_PRELOAD value does not contain slashes (is in standard search directories)
The library referenced has the set-user-ID bit enabled

LINUX
CNO
Programming

Lab 4

LD_PRELOAD

ManTech

Tasks

- Given the executable `patience`, obtain a password
  - The program will output the password in 3 hours
  - (You won’t be given 3 hours to work on this problem)
- Tips:
  - `ltrace` and `strace` can help you understand what a program is doing
    - `ltrace` traces library calls
    - `strace` traces system calls
  - `objdump -T` identifies loaded dynamic symbols

Lab Results

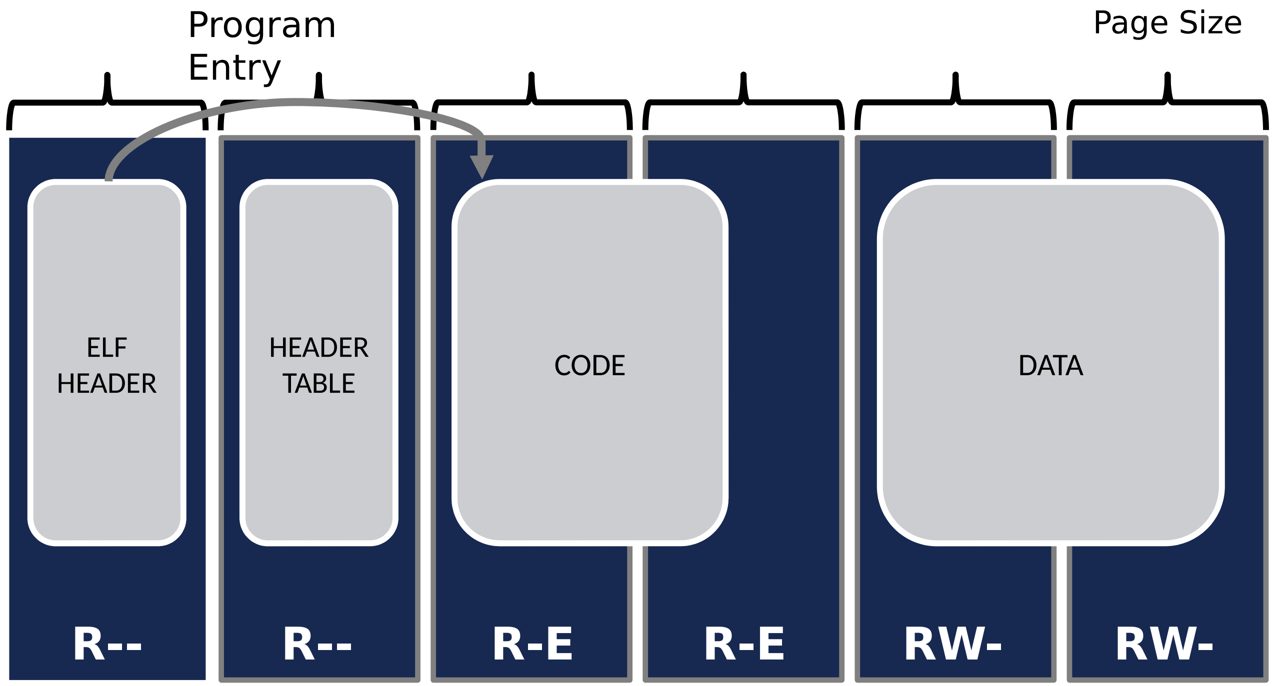
- LD_PRELOAD is used in a lot of userland rootkits and tools, as it is relatively simple to replace calls
- To find the original symbol
  - `dlsym(RTLD_NEXT, "symbol");`

Injection with ELF Poisoning

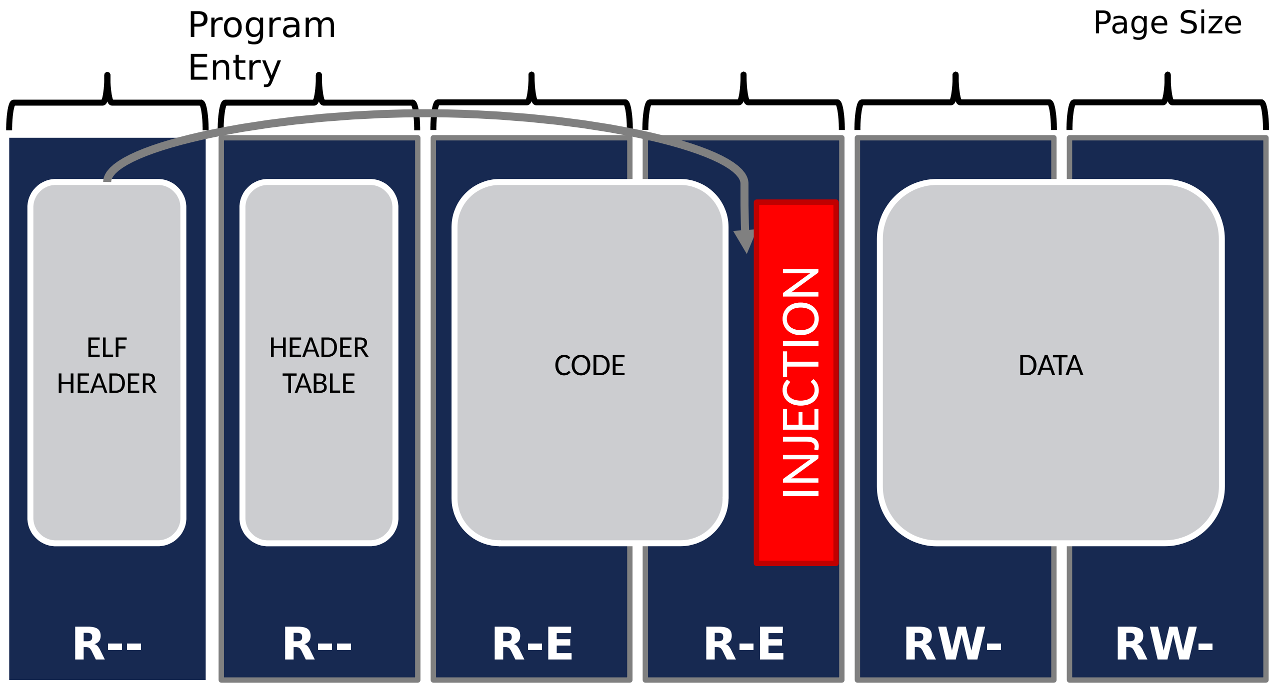
ACTP
ADVANCED CYBER TRAINING PROGRAM

- Insert code into an existing binary to run when executed
- Point the ELF Entry point to your code, before original
- Where to put the code?
  - Re-link the ELF together, insert an executable section
  - Hijack an existing section that isn't needed
    - `.note`
  - Hijack existing headers for a section that isn't needed
    - `.note`
  - Fit in the cracks
    - Sections that share the same permissions are mapped in page-sized chunks
    - The entire executable area might have some space left at the end

ELF Layout


Program
Entry
Page Size
ELF
HEADER
HEADER
TABLE
CODE
DATA
R--
R--
R-E
R-E
RW-
RW-

ELF Layout


Program
Entry
Page Size
ELF
HEADER
HEADER
TABLE
CODE
INJECTION
DATA
R--
R--
R-E
R-E
RW-
RW-

Injection with ELF Poisoning

Pros

- Simple
- Injected Code semi-hidden
    - objdump only shows section, not extra area afterward.

Cons

- Changes file bytes
    - Changes Hash digest
- Might not exist enough space left over

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 5

Poisoned ELF

Tasks

ACTP
ADVANCED CYBER TRAINING PROGRAM

- Given some boilerplate code, fill out `infector.c` and `payload.S`.

Infect a copy of `ssh` so that it will print out “Payload injected” and continue

  - Optionally cause it to connect to evil.com, no matter what site they specified to connect to

- Tips:
  - Look at `/user/include/elf.h`
  - Program headers define the segments
    - These are what you look at to find gap space, not section headers
  - If you modify the arguments to the original
    - You can get away with a `NULL` environment pointer
    - **Keep in mind byte order**

ELF Poison Lab Results

- objdump does not show any injected code
- File size stays the same
- File hash changes

Injection with Proc Filesystem

- `/proc/<pid>/mem`
  - With sufficient permissions (see man proc) one can read and write to another processes memory
- `/proc/<pid>/maps`
  - Shows currently mapped memory regions and access permissions
- You can read, write, and you know where everything is
  - Have fun!
- Permissions depend on how ptrace permissions are set up
  - PTRACE_MODE_ATTACH_FSCREDS check for `/mem`
  - PTRACE_MODE_READ_FSCREDS check for `/maps`
- By default on many distros, this means same user only

Injection with /proc/

- Pros
  - Simple
  - Language agnostic
  - Works with already executing binaries
- Cons
  - Must be able to debug the binary anyways
  - Can require some kind of synchronization to prevent target from changing memory and layout unexpectedly

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 6

Using the proc filesystem to alter execution

Tasks

- The program `auth` has been provided. Without altering the code, cause the `dowin()` function to execute
- Tips:
  - You can replace return addresses on the stack
  - `auth` is compiled as a `PIE`. You won't be able to use hardcoded addresses

Lab Results

- `/proc/<pid>/maps` provides easy ASLR defeats, if you can access it
- Make sure the program is in a good waiting state before you change anything on the stack

CreateRemoteThread in Linux

- Windows was built for thread injection, with its `CreateRemoteThread`/`CreateRemoteThreadEx` functions
- `CreateRemoteThread` allows the calling process to inject and run code of its choosing into another process
- Linux has no true equivalent
  - But it has a nice debugging API
- Using the `ptrace` system call, we can make our own `CreateRemoteThread` equivalent

Ptrace Abilities

- `gdb`, `strace`, and `ltrace` all use `ptrace` under the hood
- Allows for inspection and alteration of a running program
  - Get alerted when the process is signaled
  - Read / Write memory in process
  - Read / Write registers
- Also can be used to create a sandbox applications

Injection with Ptrace

- `ptrace` is a system call. (101 sys_ptrace)

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

- Different `ptrace` requests used to control program
  - PTRACE_ATTACH
    - Sends `SIGSTOP` to the thread, and attaches the tracer
  - PTRACE_PEEKTEXT, PTRACE_PEEKDATA
    - Used to read `sizeof(long)` bytes from the tracee
  - PTRACE_POKETEXT, PTRACE_POKEDATA
    - Used to write `sizeof(long)` bytes to the tracee
  - PTRACE_GETREGS, PTRACE_SETREGS
    - Gets and sets registers on the tracee

Injection with Ptrace *(continued)*

- `PTRACE_CONT`
  - Continues execution
    - To resynchronize, use `waitpid` to wait on the tracee to receive a signal
    - Insert `0xCC` as a breakpoint, then wait until `SIGTRAP`
- To inject a .so library:
  - `PTRACE_ATTACH`
    - Wait until tracee receives `SIGTRAP`
  - Find Safe spot to clobber in `.text` section
    - glibc’s entry is never used, won’t cause problems
  - Write in code to map memory for path string and new stack
  - Change registers + for call, and continue
  - Write in code to call `dlopen`
  - Restore, cleanup, and detach

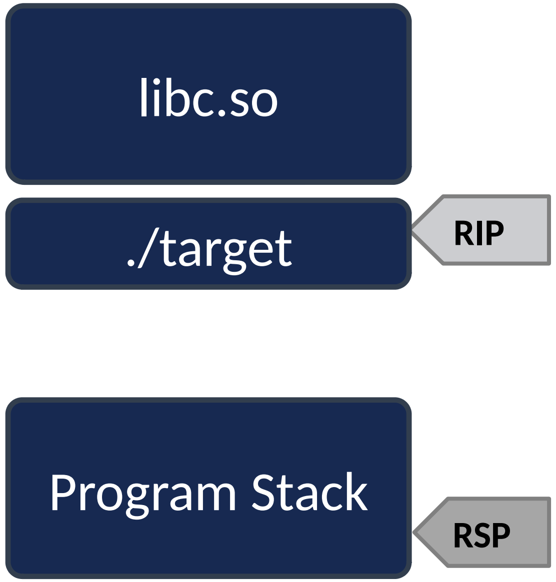
Injection Strategy

Tracer Actions

STEP 1

- Attach to target
  - PTRACE_ATTACH
    - `waitpid` until attached
    - will stop the program running
- Save State
  - PTRACE_PEEKTEXT
    - What we will clobber later
  - PTRACE_GETREGS
    - What we will change later

Tracee Memory



Injection Strategy

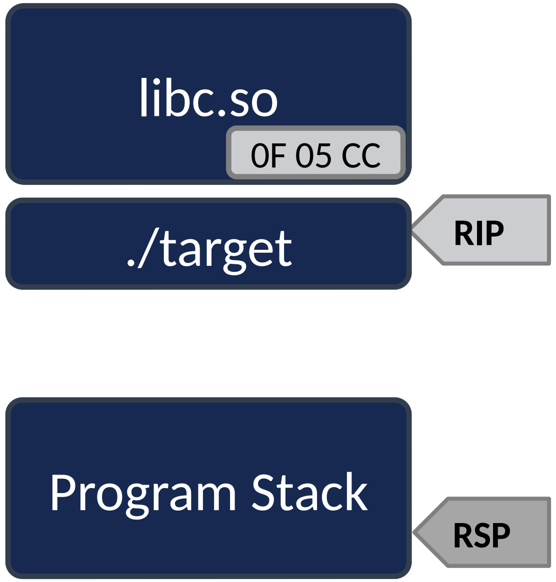
ACTP
ADVANCED CYBER TRAINING PROGRAM

Tracer Actions

STEP 2

- Inject System Call opcodes
    - PTRACE_POKETEXT
        - glibc unused `e_entry`
            - Should be safe always
            - Should exist always
        - `0x0F 0x05 0xCC`
            - `syscall; int3`

Tracee Memory



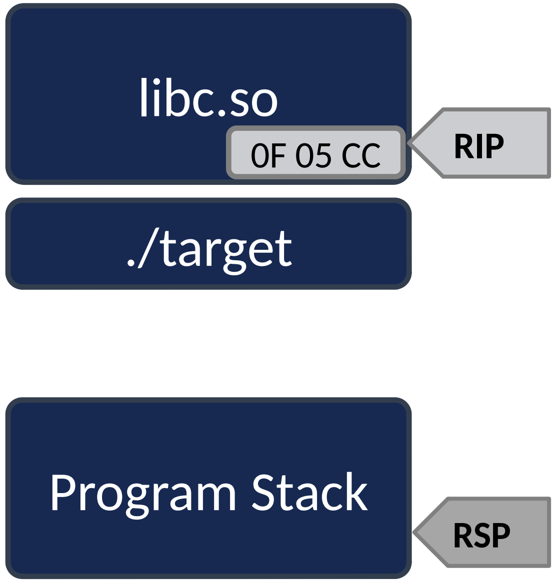
Injection Strategy

Tracer Actions

STEP 3

- Set registers for System Call
  - PTRACE_SETREGS
    - `%rip` -> glibc `e_entry`
    - `%rax` -> mmap syscall number
    - mmap arguments
      - See syscall table

Tracee Memory



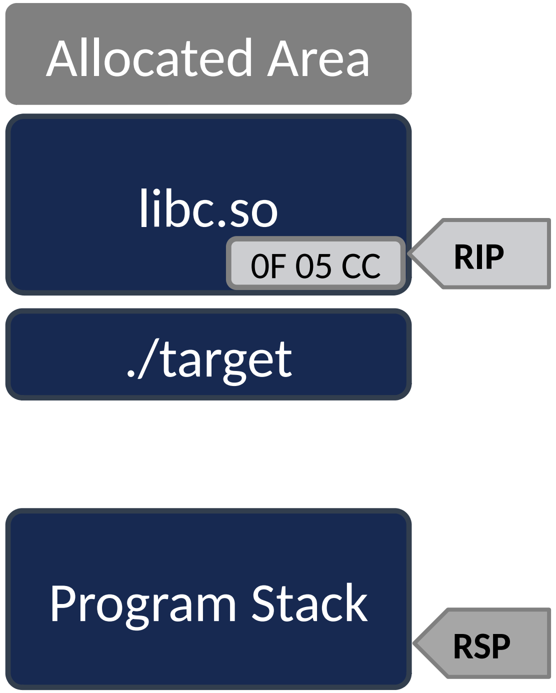
Injection Strategy

Tracer Actions

STEP 4

- Run the System Call
  - `PTRACE_CONT`
    - Let our system call happen
  - `waitpid`
    - Should be alerted when `0xCC` hit (breakpoint)
    - Ignore other signals

Tracee Memory



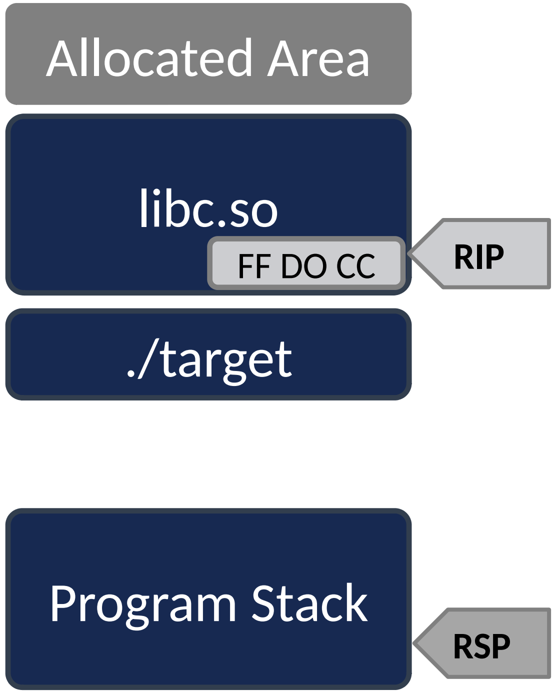
Injection Strategy

Tracer Actions

STEP 5

- Inject indirect call opcode
  - PTRACE_POKETEXT
    - glibc unused `e_entry`
    - `0xFF 0xD0 0xCC`
      - `call %rax; int3`

Tracee Memory



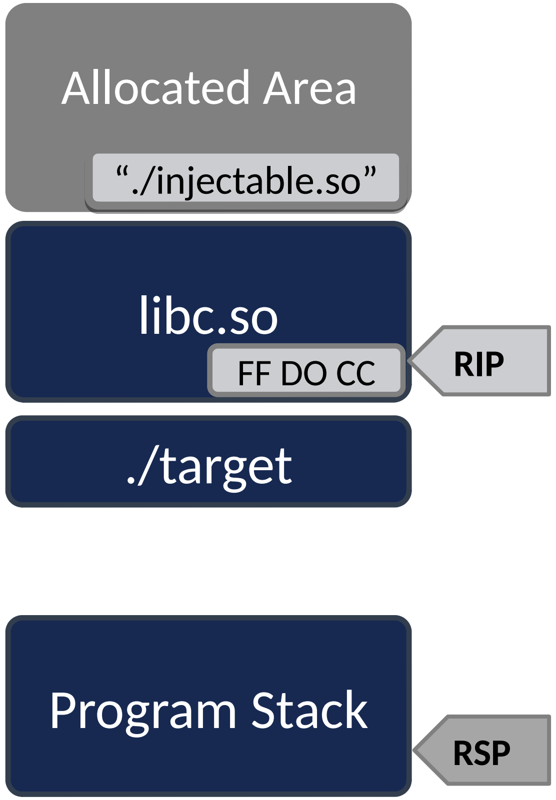
Injection Strategy

Tracer Actions

STEP 6

- Write in .so path argument
  - PTRACE_POKETEXT
    - .so path put into newly Allocated Area

Tracee Memory



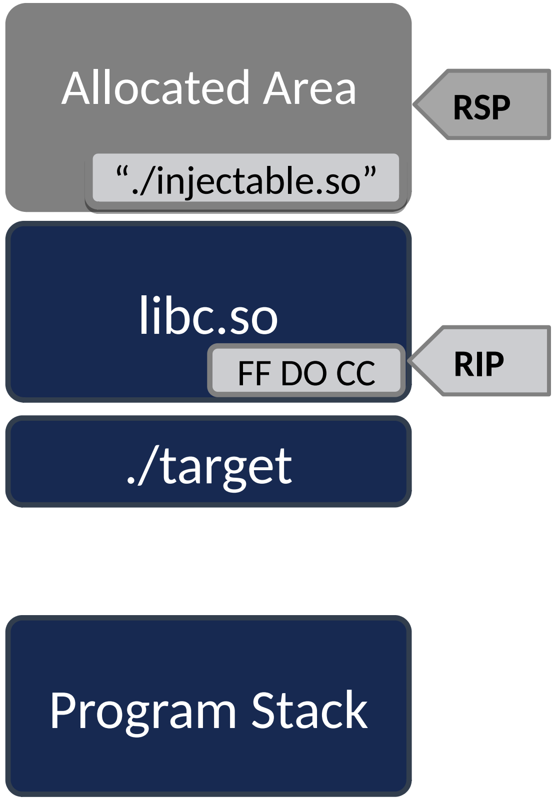
Injection Strategy

Tracer Actions

STEP 7

- Set Registers for `dlopen` call
  - PTRACE_SETREGS
    - `%rip` -> glibc `e_entry`
    - `%rax` -> `dlopen` address
    - `%rsp` -> Allocated Stack
    - `dlopen` arguments
      - `%rdi` -> .so path
      - `%rsi` -> RTLD_LAZY

Tracee Memory



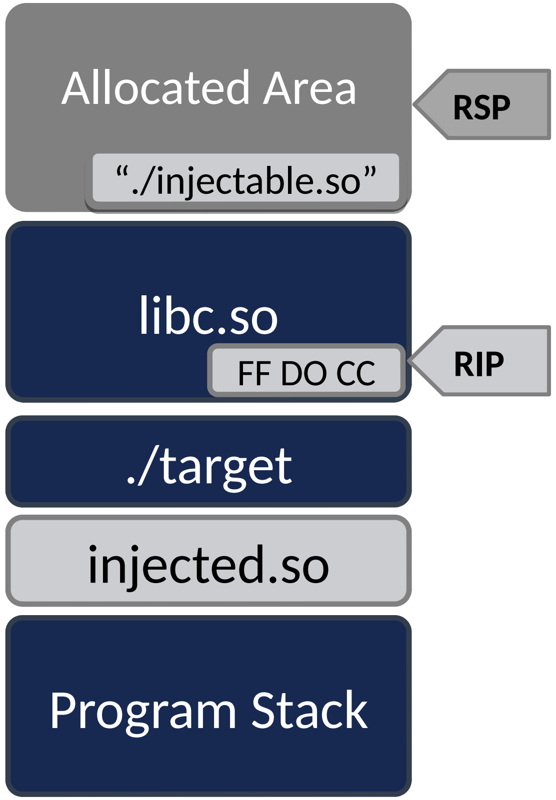
Injection Strategy

Tracer Actions

STEP 8

- Run the dlopen call
  - PTRACE_CONT
    - Cause the .so to load
    - .so constructor runs
  - waitpid
    - Wait again for breakpoint

Tracee Memory



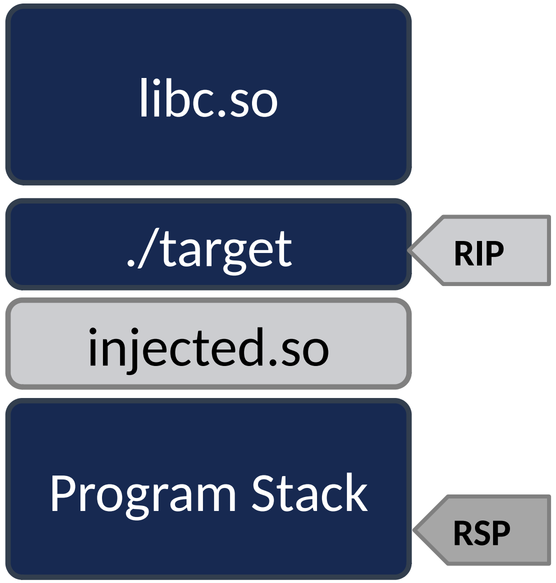
Injection Strategy

Tracer Actions

STEP 9

- Cleanup…
  - `munmap`
  - un-clobber glibc `e_entry`
  - put registers back
  - `PTRACE_DETACH`
    - Should continue program
    - Make sure not to set `PTRACE_O_EXITKILL`, or it will kill the process as well

Tracee Memory



LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 7

CreateRemoteThread on Linux

Tasks

- `setuid` program `whosthebest` forks off an unprivileged process
  - Unprivileged process sends commands to privileged process
- Use `ptrace` system call to inject dangerous commands from the unprivileged process

Ptrace Injection Lab Results

- Why couldn't we attach to the main process?
- Lots of other similar approaches out there, with lots of similar mistakes:
  - Not thread safe
    - Clobber used code
  - Don't properly wait on signals
  - Don't properly find object in memory

- How could you use this technique without a .so file on disk?
  - `memfd_create`

ADVANCED CYBER TRAINING PROGRAM

LINUX
CNO
Programming

Hooking

ACTP
ADVANCED CYBER TRAINING PROGRAM

Enabling Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe the purpose of hooking
- List different types of hooks
- Describe best practices for hooking
- Describe various hook detection techniques
- Implement basic code patching
- Implement entry stub trampolines

Introduction

- What is hooking?
  - Methods by which you can divert execution to your code instead of or in addition to existing code (usually OS code)
- Why hook?
  - Notification
  - Input/Output Filtering
  - Patch
  - Replace Functionality
  - Deny Access
  - Circumvent Security
- Examples?

Dynamic Linking vs Static Linking

- Static linking- archive files (.a)
  - All code required is compiled into the executable
  - Executables are large
  - Change in library requires recompilation of executable
  - Fast
- Dynamic linking- shared object files (.so)
  - Program determines location of code at runtime
  - Requires .so file to be present
  - Some overhead associated with dynamic linking process
  - Allows easy reuse of code
  - Allows for updating of .so without recompilation

Relevant Sections

- `.plt` - Procedure Linkage Table (PLT)
- `.got` - Global Offset Table (GOT)
- `.got.plt` - Section of GOT used by PLT
- `.dynamic` - Holds dynamic linking information
- `.rela.dyn` - runtime/dynamic linking table
  - Holds information about variables that must be relocated when the binary is loaded
- `.rela.plt` - runtime/dynamic linking table
  - Holds relocation information about functions used

Global Offset Table

- Global Offset Table
  - Table of addresses stored in the data section
    - Helps us solve the problem of absolute addresses in PIC code
  - Used to find the addresses of global symbols at runtime
    - Unknown at compile time
  - Three reserved entries
    - GOT[0] holds the address of the `.dynamic` section
      - Used by dynamic linker (`ld-linux.so`) to find all information later needed for runtime relocation and dynamic linking
    - GOT[1] holds the address of the link_map structure that contains information about the dynamic linker
    - GOT[2] holds the address of the dynamic linker code
  - Afterwards, one entry per global symbol

Procedure Linkage Table

- PLT- Procedure Linkage Table
  - Contains trampolines for code defined in dynamic libraries
    - Solves problem of execution transfer in PIC code
- When a function from a shared object is called, the PLT entry is actually called

```
0000000000401136 <main>:
  401136:       55                      push   %rbp
  401137:       48 89 e5                mov    %rsp,%rbp
  40113a:       48 83 ec 10             sub    $0x10,%rsp
  40113e:       be 01 00 00 00          mov    $0x1,%esi
  401143:       bf 10 20 40 00          mov    $0x402010,%edi
  401148:       b8 00 00 00 00          mov    $0x0,%eax
  40114d:       e8 de fe ff ff          callq  401030 <printf@plt>
  401152:       be 02 00 00 00          mov    $0x2,%esi
```

Procedure Linkage Table *(continued)*

- The PLT entry for each function first jumps to the address specified in the corresponding GOT (`.got.plt`) entry
- If the function has not been called before, this address will be the instruction right after the `.got.plt` jmp
- This code will push the index of the function to be resolved and jmp to the top of `.plt`, which contains code to perform dynamic linking

```
0000000000401030 <printf@plt>:
  401030:       ff 25 e2 2f 00 00       jmpq   *0x2fe2(%rip)        # 404018 <printf@GLIBC_2.2.5>
  401036:       68 00 00 00 00          pushq  $0x0
  40103b:       e9 e0 ff ff ff          jmpq   401020 <.plt>

0000000000401040 <malloc@plt>:
  401040:       ff 25 da 2f 00 00       jmpq   *0x2fda(%rip)        # 404020 <malloc@GLIBC_2.2.5>
  401046:       68 01 00 00 00          pushq  $0x1
  40104b:       e9 d0 ff ff ff          jmpq   401020 <.plt>
```

Procedure Linkage Table *(continued)*

- Linking is performed by:
  - pushing GOT[1] - the link_map object with references/information required by the dynamic linker
  - jumping to GOT[2] - the dynamic linking code

```
Disassembly of section .plt:

0000000000401020 <.plt>:
  401020:       ff 35 e2 2f 00 00       pushq  0x2fe2(%rip)        # 404008 <_GLOBAL_OFFSET_TABLE_+0x8>
  401026:       ff 25 e4 2f 00 00       jmpq   *0x2fe4(%rip)        # 404010 <_GLOBAL_OFFSET_TABLE_+0x10>
  40102c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000401030 <printf@plt>:
  401030:       ff 25 e2 2f 00 00       jmpq   *0x2fe2(%rip)        # 404018 <printf@GLIBC_2.2.5>
  401036:       68 00 00 00 00          pushq  $0x0
  40103b:       e9 e0 ff ff ff          jmpq   401020 <.plt>
```

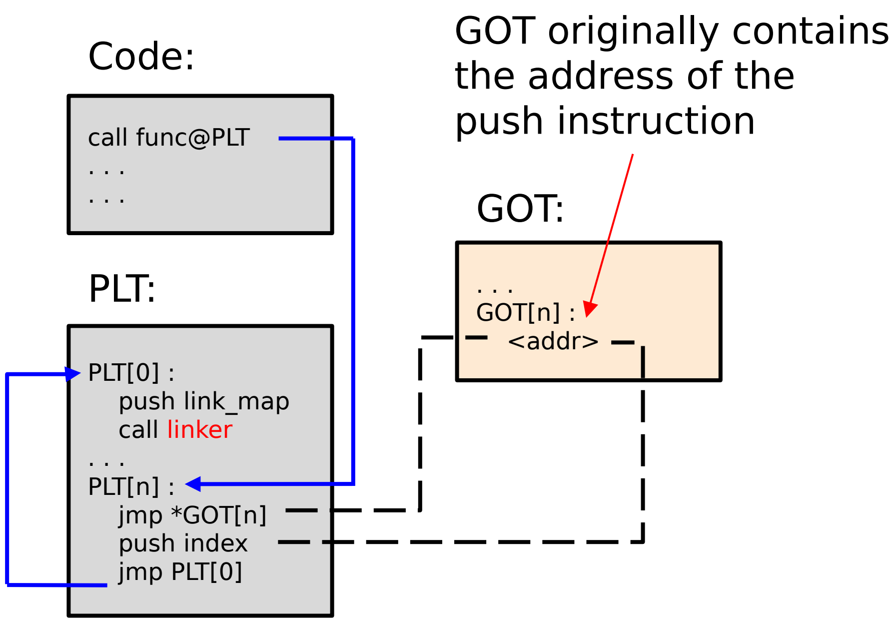
- The linker will place the address of the function in the GOT and then call it
- The function's address will already be in the GOT for future calls

.got vs .got.plt

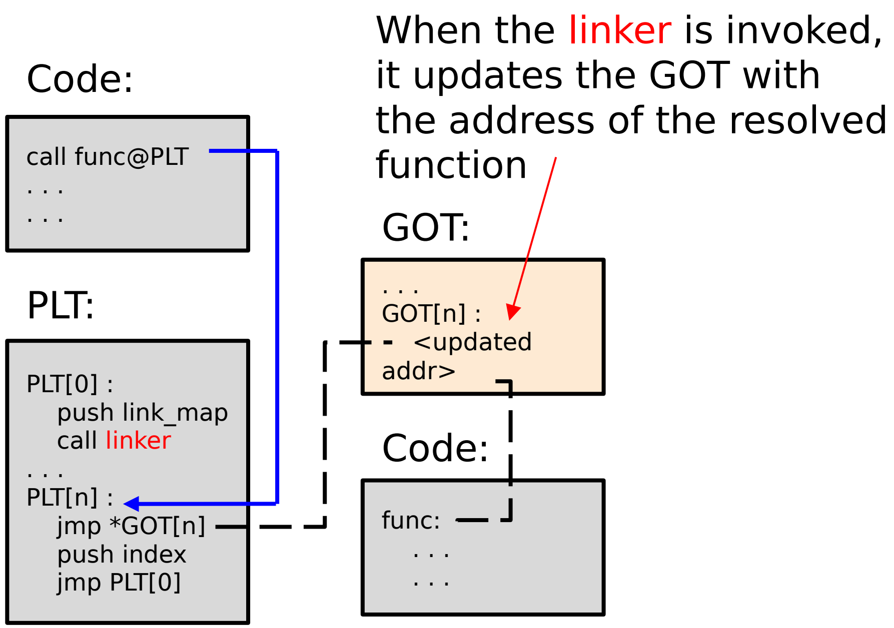
- `.got.plt` works in conjunction with PLT to help resolve **functions** from shared objects
  - More information about each of these functions is contained in the `.rela.plt` section
- `.got` is for addresses of global **variables** that are relocated when binaries are **loaded**
  - The variables that must be relocated for each binary are listed in that binary's `.rela.dyn`
- Thus, entries in the `.got` must be resolved immediately (when the binary is loaded) while entries in `.got.plt` can use "lazy binding" (be resolved right before they are executed in the code)

Summary

First call (before dynamic linking)



Subsequent calls (after dynamic linking)



LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 8

Got hooks?

Lab Description

- A target binary with setuid is performing some secret communication using `send`
- The binary will attempt to load a debugging library, if it finds one on the system
- Inspect the binary to see what you need to do to get it to load the debug library, and then create your own library
- Your library should hook `send` and `printf` in the target binary by overwriting the appropriate pointers in the global offset table
- Using your hooks, add a prefix to the **`printf`** output and display the data being sent with **`send`**
- Note that `LD_PRELOAD` does not work on setuid binaries

Lab Description

- It may be useful to break down this lab into the following steps:
  - Locate the binary's ELF image in memory (see /proc/self/maps)
  - Parse the portion of the ELF in memory to find the GOT
    - Use /usr/include/elf.h as a guide for these substeps
    - Find the program header of type `PT_DYNAMIC` and get an address to the section
    - Iterate through the dynamic section until you find one with a tag of `DT_PLTGOT`
    - Return a pointer to the GOT (see `d_ptr` field)
  - Hook the GOT entries
    - Iterate through the GOT to find addresses of the functions you want to hook
    - If they're not there, try again later (feel free to use a thread to periodically attempt hooking)
    - Why might you not find the address of a function in the GOT?
  - Create your wrapper functions to exfiltrate information

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Pointer Replacement

ManTech

Overview

- Compiled code calls external functions by using (a table of) function pointers
  - As opposed to relative offsets
  - What are some examples of this?
- Hook by changing the value of the function pointer to point some other code
- Save the original pointer to call the original function
- One of the simpler forms of hooking

Examples

- GOT replacement
  - By replacing the GOT entry pointer for a function, you can **usually** hook that function just for that module
  - Will not hook explicit imports
  - Will not work if import address stored off elsewhere
- vDSO overwrite

vsyscall

- The precursor to vDSO is the “vsyscall” mechanism
  - System calls have overhead associated with kernel context switch
  - Can reduce this by mapping information required from the kernel and a quick implementation of the syscall into user memory
    - Example- `gettimeofday`
  - Problems
    - The vsyscall page could only hold four entries
    - The vsyscall page had to be statically mapped to the same location in memory in all processes

vsyscall *(continued)*

- Mitigation
    - Remove useful instructions from vsyscall page
    - Move variables into other pages with execute permissions turned off
    - Replace remaining code with trap instructions
        - These will trap into the kernel and emulate the vsyscall
        - Finally producing a kernel system call emulating a vsyscall which was put there to speed up that same system call

vDSO

- vDSO = virtual dynamic shared object
  - Mapped into every user-mode process
  - vsyscall without the limitations
  - Kernel functionality is still exposed in userspace, but
    - Memory is allocated dynamically
    - We have room for more than four entries
- Find location using
  - `#include <sys/auxv.h>`
  - `getauxval(AT_SYSINFO_EHDR);`
- Depending on kernel configuration, vDSO may be either RX or RO, can use **`mprotect`** to overwrite

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Callback Registration

Overview

- Sometimes you can register a hook through an API
- Examples:
  - Signal handlers
  - Thread-local storage
  - Constructors (**`.init`**) and destructors (**`.fini`**)
  - **`atexit`**

Callback Registration Issues

- **`atexit`**
  - Is not called if program calls exec()
  - Is not called if process terminates abnormally due to delivery of a signal
- Limited to the information and filtering provided by the API

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Code Patching

Overview

- Code patching is overwriting machine instructions to modify function behavior
- This is the most powerful, but most complicated, form of hooking
- Usually consists of a patch to *divert* execution to another function
- Sometimes the patch itself changes execution behavior
  - Example: return immediately with a success value

Basic Diversion

- Any sequence of instructions that redirects execution is an acceptable patch
- Typically placed at beginning of a function
- Rarely placed mid-function
  - Would need to be tailored specifically for the function
- The 5-byte relative jump (`0xE9`) is common if you only need to jump < 2GB
- Otherwise:

```
movq $0x9f44444444, %rax
jmp *%rax
```

Alternative Patches

Other than diverting execution:

- Disable function
    - Always return either success or an error
    - Can also disable specific case
- Replace basic blocks or an entire function body

Entry Stub Trampolines

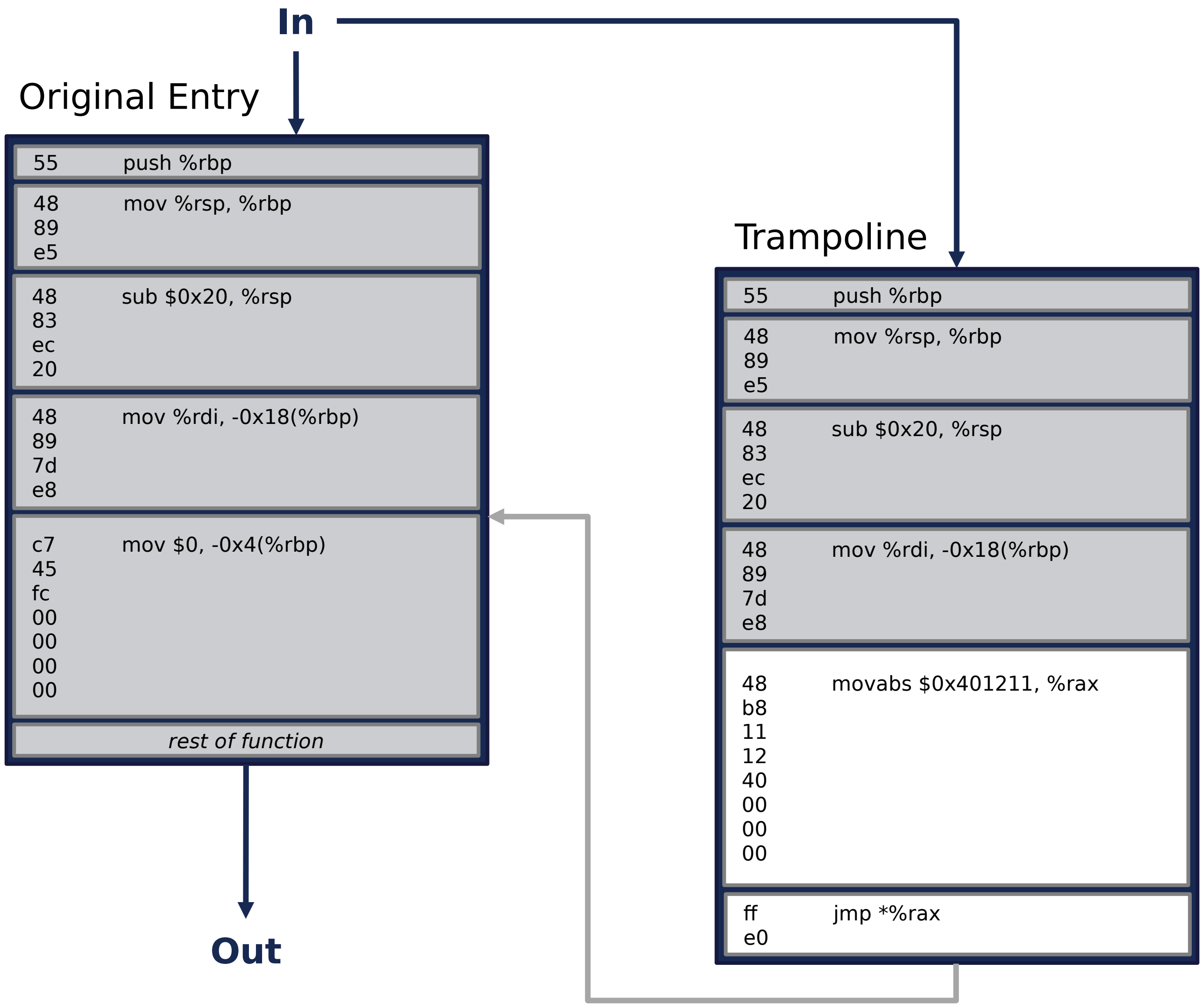
- Often you are hooking to wrap the original function
- But the hook modified the original instructions
  - What happens if you simply call the original function?
- How can you call the original function?

Entry Stub Trampolines *(continued)*

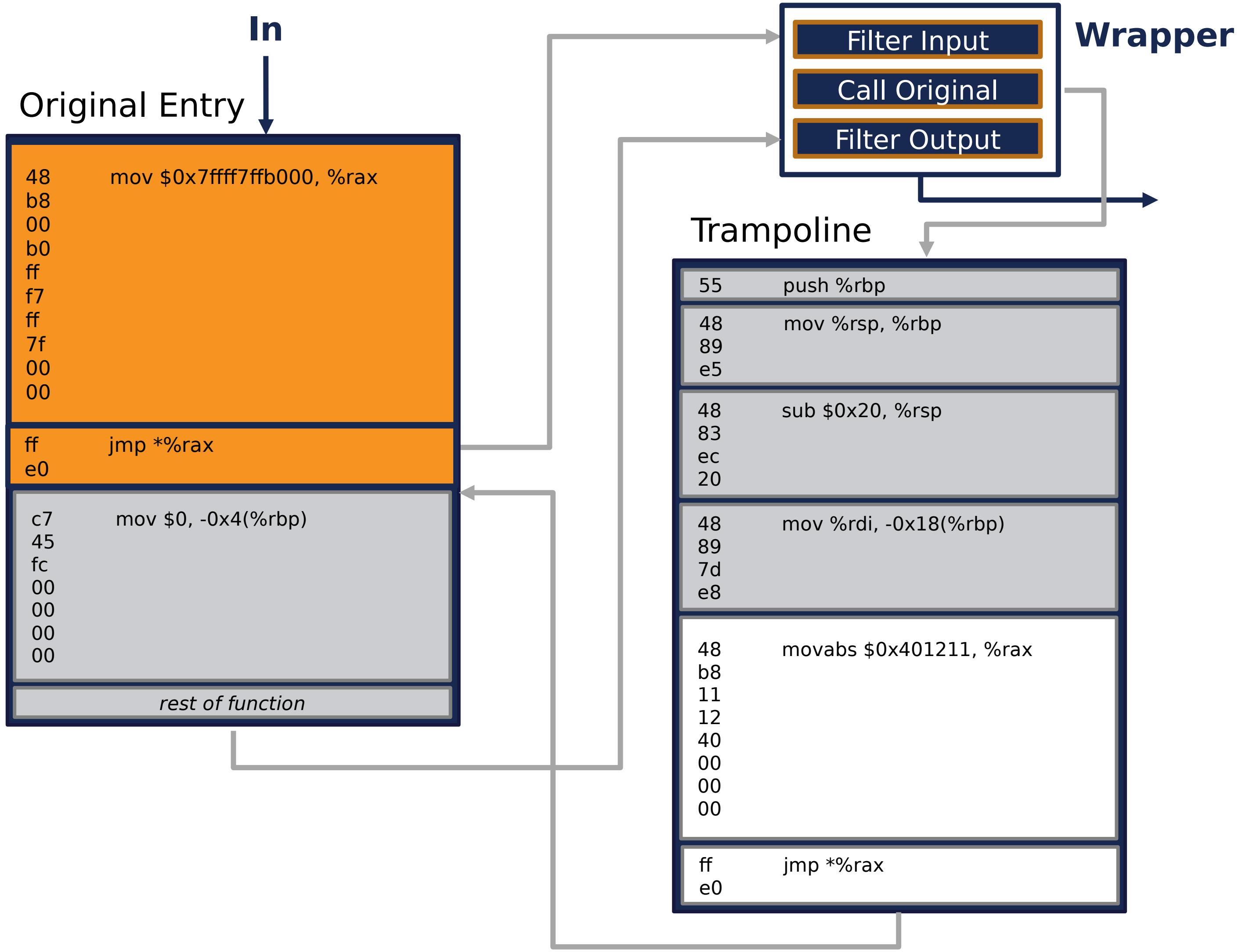
- The machine instructions overwritten by the patch are saved off in a “trampoline”
- The end of this trampoline jumps back to the original function, after the patch
- Calling this stub will therefore emulate the original function*
  - What problems can occur when executing the saved instructions?

*This assumes the original function can support relocation of those instructions. For example, if there was a block end or relative offset within the patched instructions, this would not work. But, when you are talking about compiler-generated function entries, this usually is fine

Entry Stub Trampolines *(continued)*



Entry Stub Trampolines *(continued)*



Entry Stub Trampolines *(continued)*

- The patch may overwrite only part of an original instruction
- You need a ***length***-disassembler to know:
  - Where to jump to at the end of the trampoline
  - How many bytes to put in the trampoline
  - **ndisasm**, **udis86**, or **objdump** are handy
  - A full disassembler is not needed

Multi-Stage Patches

- What is a multi-stage patch?
  - Group of more than one patch that diverts execution in a chain
  - Adds “safety instructions” between first patch and hook code
- Why might you use one?
  - Avoid heuristic hook finders
    - “safety instructions” means jumps to areas within the target module

LINUX
CNO
Programming

Lab 9

Stub your toe.

Who'da thunk?

Tasks

- In this lab, you will implement basic code patching
- Implement the following functions:
  - `patch_code`
  - `write_absolute_jump`

Tasks *(continued)*

- Implement **`patch_code`**
- It is a good idea to have all of your code patching use a single support function
  - This gives you one single place to put most of your safeguards
- Arguments:
  - *`*target`*: address to apply patch
  - *`*patch`*: points to bytes to use as patch
  - *`len`*: number of bytes in patch
- The **`patch_code`** function will write the bytes to the address

Tasks *(continued)*

- The safety features in **`patch_code`**:
  - Exception handler wraps all code, to help protect against bad pointers or unexpected problems
  - **`mprotect`** call ensures that code is made writeable before applying patch
  - **`mprotect`** call reapplies original permissions after patch is applied
    - Not strictly necessary but makes patched code look “normal”

Tasks *(continued)*

- The safety features **NOT** in **`patch_code`**:
  - There is no attempt to make an atomic write
    - `__atomic_exchange_n`
    - Assembly (`cmpxchg` or `cmpxchg8`)
    - You may implement this feature if you like
  - No attempt to gain higher execution priority
    - Increasing thread priority
    - Entering critical section

Tasks *(continued)*

- Implement **`write_absolute_jump`**
- The **`write_absolute_jump`** function writes a relative jump instruction at `jump_from` that will redirect execution to the target, `jump_to`
    - Offset calculation:
        - offset = (target – (source + <size of patch>))
- It uses **`patch_code`** to apply the jump patch

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 10

Bounce me to the moon

Tasks

- In this lab, you will implement an Entry Stub Trampoline
  - Consists of:
    - A Sequence of X instructions copied from a function entry to a stub (trampoline)
    - An unconditional jump is placed at the end of the stub, which redirects execution back to the original function just after the entry hook
  - Used to call the original function when the function entry has been patched

Tasks *(continued)*

- Udis86 Disassembler
  - A full x86 disassembler implemented in C
  - For applications where space is a concern, it is recommended to have a length-only instruction disassembler
- **`injector.c`** starter code, you will see there is a function (implemented using Udis86) to calculate the length of an instruction

```
unsigned int get_instruction_length(uint8_t* addr);
```

Tasks *(continued)*

- **injectlib.c** also defines the `entry_stub_t` structure to hold the entry stub information:
  - *`*original_entry`*: The location of the original entry
  - *`entry_size`*: The size of the instructions stored
  - *`*trampoline`*: A pointer to executable code that will emulate the original function
    - This is `entry_size` bytes of instructions followed by a jump to offset `entry_size` bytes into the function entry

```
/* Entry Stub Trampoline structure */
typedef struct _entry_stub {
    uint8_t *original_entry;
    unsigned long entry_size;
    uint8_t *trampoline;
} entry_stub_t;
```

Tasks *(continued)*

- The *`entry_stub_create`* function allocates and initializes an entry stub based on an
  - entry point
  - the **minimum** number of bytes
- The minimum number of bytes is the size of the patch
- This function should round `size` up so that the last instruction that needs to be copied is copied completely
- After calling *`entry_stub_create`* you can call the trampoline member pointer just as you would have called the original entry

Tasks *(continued)*

- `entry_stub_hook` writes a jump patch to the original function, diverting execution to wrapper code
  - You already wrote most of this code in previous
- `entry_stub_unhook` removes a jump patch and restores the original bytes
- `entry_stub_free` frees the memory allocated for the `entry_stub_t` structure

Tasks *(continued)*

1. Update previously implemented functions:
   - `patch_code`
   - `write_absolute_jump`
2. Implement:
   - `entry_stub_create`
   - `entry_stub_hook`
   - `entry_stub_unhook`
   - `entry_stub_free`

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Hiding

Enabling Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe effective CNO hiding techniques
- Demonstrate a file hiding technique by hooking Windows API calls

- Hiding is the ability to perform your activity without raising suspicion from users or security software

Who are you hiding from?

- How you hide very much depends on who or what you are attempting to hide from:
  - Computer User
  - Anti-Virus or HIDS Software
  - Occasional Scans
  - Forensics Analyst

What Are You Hiding?

- Data
  - Files
- Activity
  - CPU Activity
  - Memory Usage
  - Disk Usage
  - User-Associated
    - e.g., key strokes
- Implant
  - Hooks
  - Patches
  - Registered Callbacks
  - Filters
  - Persistence Mechanisms
- Network Traffic
  - Connections
  - Destination
  - Content

Hiding files

- Creating a file – even some memory-backed ones, requires mapping it to the file system through some local mount point
  - Example: Adding a file to your home directory in `/home/student/myfile`
- Special files also get mapped to the file-system
  - Example: hardware devices in `/dev`
- One way to be stealthy is to avoid mapping files to obvious places in the file system
  - Using **shm_open**, you can create a memory-backed file, but it is mapped to `/dev/shm`
- More recent kernels support a better way…

memfd_create

- Since kernel version 3.17, Linux has the **`memfd_create`** system call, which allows a memory-backed file to be created that is **not associated with the file system**
- This allows us to obtain and execute remote payloads without ever touching the disk!
    - Typically, temporary files would be created in `/tmp` and executed from there – but any service monitoring file system changes would detect this
- See **`memfd_create`** man-page for more details
- Passing the path `"/proc/self/fd/X"` (where `X` is the file descriptor returned by `memfd_create`) to `execv`, `dlopen`, etc. will execute the in-memory file

LINUX
CNO
Programming

Lab 11

Can’t touch disk (dun dun dun dun)

Description

- The root user has a cronjob that executes once every minute
- The cronjob is run unprivileged and sandboxed
- The sandbox
  - Allows read access to a minimal set of system dirs
  - Monitors file creation and modification
  - Only allows files smaller than 10k to be run

Objective

- Create a stager to be run by the cronjob’s sandbox
- The stager should, once executed, should download and run a payload from a server
  - The payload is served via TCP on port 8080 from the instructor’s machine, your box, or your Fedora VM
  - When your client program connects to the server, the server will automatically send the payload
  - The server will first send 4 bytes, which indicate the size of the payload in bytes, in network (big endian) byte order

Restrictions

- The sandbox has mounted `/proc`, `/lib64`, and `/etc`, but they are read-only
- The sandbox is monitoring the file system and any attempt to open or create a typical file will result in it detecting your presence
  - This rules out operations such as `fopen` and open
- You can use **`memfd_create`**, followed by `dlopen`, to create an in-memory file, download the remote payload into that file, and then execute it
  - The payload uses a constructor function to execute automatically once it is loaded with **`dlopen`**

Bonus

- Find a way to execute the payload undetected without using **`memfd_create`**
- Think about what **`memfd_create`** and **`dlopen`** do, and recreate their basic functionality to execute the code in the payload

LINUX
CNO
Programming

ADVANCED CYBER TRAINING PROGRAM

Hardening

Inhibiting detection and analysis

Hiding

- How can we not raise suspicion?
  - Depends on who or what is looking at you
    - User
    - Anti Virus
    - Forensics Analyst
  - Depends on what is normal on target
    - If my disk spins up every time I type on my keyboard, I will get nervous
- What do we hide?
  - Data
  - Activity
    - CPU activity
    - Memory Usage
    - Disk Use
  - Implants
    - Hooks, Patches, Callbacks, etc.
  - Network Traffic
    - Connections
    - Contents
    - Destinations

How to Be Seen

- User notices something out-of-the-ordinary
- Monitoring software alerts / blocks use of sensitive API
- Scan detects implant signatures
- Inconsistent results from different sources
- Scan detects change from known good

How to Be Seen -- User Notices

- User notices something out-of-the-ordinary
  - Suspicious files
  - Suspicious processes
  - Excessive disk / CPU / Network use
  - Failures
  - Crashes

How to Be Seen -- Sensitive APIs

- Some APIs or Files may be monitored by local Antivirus
  - Startup scripts
    - `/etc/system.d`
    - `/etc/rc.*`
    - `/etc/init.*`
    - cron jobs
  - Sensitive Configurations
    - `/etc/hosts`
    - `/etc/passwd`
    - `/etc/pam.d`
  - Core binaries
    - `/usr/bin`
  - etc.

How to Be Seen -- Signatures

- Scan detects implant signatures
  - Depends on the software doing the detecting
  - Look for common patterns
  - May even emulate pieces of your program to look for signature side effects

How to Be Seen -- Inconsistency

- Inconsistent results from different sources
  - Many rootkit finders look for inconsistency
    - e.g. If the processes in `/proc` don't match the kernel's internal list
  - This works without having to know a baseline good
  - Catches those trying to hide

How to Be Seen -- Tripwire

- Scan detects change from known good
  - File System Hashes
  - Network Traffic Patterns
  - Log Patterns
- A few example Linux intrusion detection systems:
  - **Fail2ban**
    - A log based system that looks for too many password attempts, systems seeking exploits, etc
  - **OSSEC**
    - Monitors files and logs, detects rootkits, and delivers alerts and logs to a central server
  - **Suricata**
    - Monitors network traffic, with file extraction, certificate checks, etc.
  - **Tomoyo**
    - Learns system behavior via logs and other accesses, and then detects anomalies

How not to be seen

- Look like you belong
- Do things benignly
  - Only do shady stuff when you have to, and keep it brief
- "Carry a clipboard, and look busy"

What is Hardening

- Everything can be Reverse-Engineered or Detected
  - But not everything will be, or is economically feasible
- Give the reverse engineer an excuse to quit / move on to different work
- Uses:
  - Hide signatures from Antivirus
  - Hide functionality from Disassemblers / Debuggers
  - Hide the fact that you are hiding anything at all
  - Hiding proprietary functionality
  - Detecting cheating in games
  - DRM

What is Hardening

- An eternal cat-and-mouse game
  - Understanding how hardening is done allows better analysis
    - Understanding modern analysis allows better hardening
      - ...
- Lots of room for creativity
- Very specific to your situation
  - What is out of place on your target?
  - What is being scanned for?
    - What versions can you expect?
  - Do the work to know your enemy

General Hardening

- Symbols
  - Get rid of them
    - `man strip`
- `strings`
  - What patterns are in your binary?
    - Statically? At Runtime?
  - If I have "~/.ssh/id_rsa" in my binary, that is a red flag
    - How can we still use that string, but not have it show up in memory scans?
- Dynamic Library Imports
  - Do you have an ELF that imports glibc functions?
    - Gives reverse engineers good spots to hook
    - Gives propagated type information
- Run in surprising locations
  - `DT_INIT`
  - `init`
  - TLS callbacks

General Hardening – Obfuscated Strings

- Problem:
  - You want to use strings/other data in your program that is sensitive
  - The data needs to be hidden from being enumerated in the program
- Solution:
  - Hide the data
  - "Stack Strings"
    - Often seen in malware
    - Lazy solution
  - Xored Strings
  - Be Creative

General Hardening – Stack Strings

- Stack Strings
  - Again, lazy and well known technique
  - Fire Eye has a good set of IDA scripts to decode them
  - Only here as an example

```
[student@localhost ~]$ pygmentize -g ./stackstr.c
#include <stdio.h>

int main() {
        char stackstr[] = {'H','E','L','L','O',' ','W','O','R','L','D','\n',0};

        printf(stackstr);
}
[student@localhost ~]$ gcc -o stackhello ./stackstr.c
[student@localhost ~]$ ./stackhello
HELLO WORLD
[student@localhost ~]$ strings -a ./stackhello | grep -i "HELLO"
[student@localhost ~]$
```

General Hardening – Stack Strings

- Each character gets pushed
- Characters are separated by movb
- Easy to find with analysis
- Easy to see in IDA
  - 'r' on operand turns to character
- Used because simple to use
  - No crazy pre/post processing
- Watch out for optimizations
  - Could ruin stack strings
- Won't work in data section

```
0000000000401126 <main>:
  401126:       55                      push   %rbp
  401127:       48 89 e5                mov    %rsp,%rbp
  40112a:       48 83 ec 10             sub    $0x10,%rsp
  40112e:       c6 45 f3 48             movb   $0x48,-0xd(%rbp)
  401132:       c6 45 f4 45             movb   $0x45,-0xc(%rbp)
  401136:       c6 45 f5 4c             movb   $0x4c,-0xb(%rbp)
  40113a:       c6 45 f6 4c             movb   $0x4c,-0xa(%rbp)
  40113e:       c6 45 f7 4f             movb   $0x4f,-0x9(%rbp)
  401142:       c6 45 f8 20             movb   $0x20,-0x8(%rbp)
  401146:       c6 45 f9 57             movb   $0x57,-0x7(%rbp)
  40114a:       c6 45 fa 4f             movb   $0x4f,-0x6(%rbp)
  40114e:       c6 45 fb 52             movb   $0x52,-0x5(%rbp)
  401152:       c6 45 fc 4c             movb   $0x4c,-0x4(%rbp)
  401156:       c6 45 fd 44             movb   $0x44,-0x3(%rbp)
  40115a:       c6 45 fe 0a             movb   $0xa,-0x2(%rbp)
  40115e:       c6 45 ff 00             movb   $0x0,-0x1(%rbp)
  401162:       48 8d 45 f3             lea    -0xd(%rbp),%rax
  401166:       48 89 c7                mov    %rax,%rdi
  401169:       b8 00 00 00 00          mov    $0x0,%eax
  40116e:       e8 bd fe ff ff          callq  401030 <printf@plt>
  401173:       b8 00 00 00 00          mov    $0x0,%eax
  401178:       c9                      leaveq
  401179:       c3                      retq
```

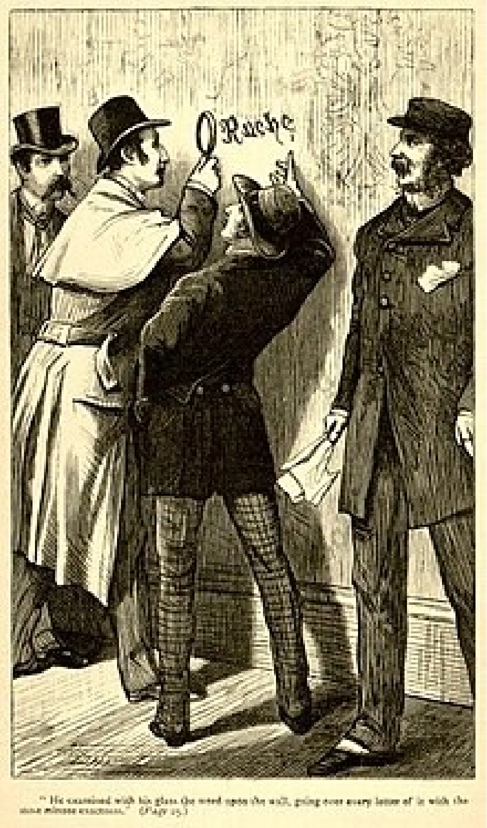
General Hardening – Obfuscated Strings

- A better approach allows compile time obfuscation, and runtime temporary deobfuscation
  - Strings start out encrypted or xored in memory
  - Translated right before use
  - Memory cleared afterward
    - Hide from memory dumps
- Can involve C Macros
  - Whatever works for your build
- Watch out for common failures
  - Make sure what you think happens at compile time actually happens at compile time
    - Fallout76 tried to have xorstrings but ended up xoring and immediately de-xoring at runtime
  - Don’t clear strings that are still in use
  - Run the C preprocessor before you run your preprocessor

General Hardening – Red Herring

ACTP
ADVANCED CYBER TRAINING PROGRAM

- Distract analyzers
- Combining readily viewable text and code can be effective in throwing off casual observers and confusing analyzers
- The more red herring code the more effective, but the cost is size
- Imply a benign reason for your code's activities
- Provide a reason for the large chunk of high-entropy data that is your real code.
  - Hash Tables
  - Compressed Images
  - Signed Certificates



Avoiding Automated Analysis – *Signature Diffusion*

ACTP
ADVANCED CYBER TRAINING PROGRAM

- Automated tools look for specific patterns and signatures
- Know what modern machine learning models see as key identifiers
  - Network packet size / timing
  - Entropy of data / code regions
  - Side effects of common techniques
- Emulate commonly used software functionality
- Never send the same payload twice
  - Use an automated tool to produce functionally equivalent code that is procedurally different
    - Different encryption methods
    - garbage/unrelated mixins
  - Use functionally equivalent instructions
  - Be creative!

Raising the Cost of Static Analysis

- To deter analysts using Static Analysis tools
  - Confuse the tool
  - Make the outcome unreasonable to understand without execution or custom processing
- Get your IDA 0-days out
  - ELF vaddr vs fileoff
    - Do they use the right one?
  - gdb used to crash if an elf had a debuglink section of size 0
- Some Techniques:
  - Packing / Compressing
  - Anti-Disassembly
  - Static Complexity
  - Be Creative

Static Analysis – Packers

- There are lots of existing packers used to:
  - Decrease Size
  - Obfuscate
- Many are available online:
    - upx is a very common open source one for Linux/ELF
  - These may trigger AV alerts
  - These often have readily available unpackers
- Heuristics can be used to detect unknown packers
  - Entropy
  - unpacking stubs
  - strange sections
  - etc.

Static Analysis – Anti-Disassembly

- Anti-disassembly: Tactics to raise the difficulty for tools and people to statically analyze your code
- Overlapping instructions
  - In x86, we have variable sized instructions
  - Execute an instruction at an offset within a previously executed instruction
    - Disassemblers have a really hard time with this
    - Even if the tool can follow it, which instruction do they show to the analyst?

Static Analysis – Anti-Disassembly

- Even just a bad opcode put in the middle of a function can trick mess up a few lines for disassemblers

```
char stackstr[] = {'H','E','L','L','O',' ','W','O','R','L','D','\n',0};
__asm__ (
     "jmp PAST_GARBAGE_BYTE\n"
     ".byte 0xe9\n"
     "PAST_GARBAGE_BYTE:\n"
     :::
);
printf(stackstr);
```

```
(With Symbols)
401162:    eb 01            jmp    401165
<PAST_GARBAGE>
401164:    e9               .byte 0xe9
<PAST_GARBAGE>:
401165:    48 8d 45 f3      lea    -0xd(%rbp),%rax
401169:    48 89 c7         mov    %rax,%rdi
40116c:    b8 00 00 00 00   mov    $0x0,%eax
401171:    e8 ba fe ff ff   callq  401030
<printf@plt>
```

```
(Stripped)
401162:    eb 01            jmp    401165
401164:    e9 48 8d 45 f3   jmpq
ffffffff3859eb1
401169:    48 89 c7         mov    %rax,%rdi
40116c:    b8 00 00 00 00   mov    $0x0,%eax
401171:    e8 ba fe ff ff   callq  401030
<printf@plt>
```

Static Analysis – Anti-Disassembly

- Indirect Code
    - Mutable Function Tables / Function pointers
    - Signal Handlers
    - call / jump redirection at runtime
  - Takes longer to trace
  - Makes purely static analysis very difficult
- Conditional Calling Conventions
- Equivalent instructions
  - Movfuscator
    - fun compiler that "compiles a program into 'mov' instructions, and only 'mov' instructions."
- Be Creative

Deterring Dynamic Analysis

- What do you do if you are actually being debugged?
  - Detect that you are being debugged and don't do the thing
    - Clear registers / stack and jump to random?
    - Do something else?
    - Whistle nonchalantly
  - Make the analyst's life terrible / be annoying
    - Whistle aggressively

Dynamic Analysis – Break Points

- Software Breakpoints actually change code
  - `int 3`
    - Opcode `CC` or `CD 03` (both are a valid `int 3`)
  - Detect breakpoints and hooks by checksumming yourself in a separate thread / process
- Hardware Breakpoints don't change anything
  - gdb watchpoint
  - Limited number of hardware breakpoints available (actual registers on the cpu)
- False Software Breakpoints
  - Have a `SIGTRAP` handler, and send out multiple `int 3` that they are not expecting
- Generate Single Step exception
  - `EFLAGS` or `ICEPB` (`int 1`)
  - Your handler will not be run if a debugger is attached
- Check timing to find presence of debug break

Dynamic Analysis – Ptrace

- Processes cannot be `ptrace` attached to twice
  - In another process try to attach to your main process, see if it fails
    - Use `PTRACE_O_EXITKILL` so they can't just kill the child and attach
    - You could also try `PTRACE_TRACEME` from the same process
    - Use the `ptrace` syscall, don't get intercepted
    - This check can be defeated by hooking ptrace
- Look in `/proc/<pid>/status`
  - Look at `TracerPID`
- Use the `CLONE_UNTRACED` flag to spawn a untraced child

Dynamic Analysis – LD_PRELOAD

- Detect `LD_PRELOAD` by looking for the `LD_PRELOAD` variable in your `environ`
  - If they hook soon enough, though, they can try to remove it before you see
  - They can also preload without the environment variable
    - `/etc/ld.so.preload`
    - `LD_LIBRARY_PATH`
- Exec an image that isn't preloaded
  - Clear the environment variable, if that was the method used
  - Look in `/proc/self/maps` for unexpected images

Dynamic Analysis – Process Dumps

- Programs like `gcore` can produce core dumps of running programs.
  - Capture the unpacked /unencrypted data
  - `gcore` doesn't work if process is already being traced
  - `gcore` also doesn't dump memory marked don't dump
    - `MADV_DONTDUMP` flag with `madvise`
    - But generating a core by sending an unexpected signal will still dump the memory
  - The kernel won't take a core dump if the process is not dumpable
    - `PR_SET_DUMPABLE` flag with `prctl`

Dynamic Analysis – Analysis Environment

ACTP
ADVANCED CYBER TRAINING PROGRAM

- Another way to avoid analysis is to check your environment
- Detect Debuggers open
- VM detection
    - system uptime
    - number of cores
    - special vm only instructions
    - loaded libraries
    - device IDs and names
        - `dmsg`
        - `dmidecode`
    - Get Creative
- Emulator Detection
    - Do things that emulators have a hard time keeping up with

Dynamic Analysis – Side Effects

- The best analysis doesn't care about your obfuscation
- Even if you are running out of a custom interpreter for a custom bytecode that would take a very long time to reverse, all that matters is side effects
- Good analysts look at what you are doing, regardless of how you do it
- Be mindful of what effects you are having on the machine, and in what situations they are acceptable

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 12

Hide and Seek

Obfuscate and Contemplate

Tasks

- Objectives
  - Take some provided starter code and obfuscate it for a given amount of time
  - Then, swap executables with a class member, and try and obtain the other's flag
- Rules
  - Don't do anything malicious (booby traps, fork bombs, delete files, etc.)
  - The steps taken to get the flag from the unobfuscated starter code must also retrieve the flag from the obfuscated binary
  - You will be given a number, and should only look in the folder with that number for your starter code
    - Or do what you want, I am a bullet point not a cop
- Bonus
  - Share your binary around, and get as many flags as you can

LINUX CNO Programming

ADVANCED CYBER TRAINING PROGRAM

Lab 13

Packer

Tasks

ADVANCED CYBER TRAINING PROGRAM

- Make a very simple packer that can take arbitrary executables as input
- The output of your packer should be runnable and should behave identically to the original input binary
- Be creative!

Tasks *(continued)*

- Tips:
  - **`xxd -i <file> >> out.c`** will convert the input file <file> into bytes in C array syntax and append it to out.c

LINUX
CNO
Programming

ACTP
ADVANCED CYBER TRAINING PROGRAM

Lab 14

Avoid Detection

 ManTech

Tasks

- You are given the source to a simple program that gathers needed machine-specific information
- Modify the program to be able to run without being detected by the antivirus
  - For our use, the antivirus is what we use to run our program
    - `./antivirus –t ./infoFinder`
- You must retain the original functionality of the `infoFinder` program, all in the one ELF file
- Bonus:
  - Reverse engineer the antivirus to find any hidden functionality that can help to bypass it
  - Be able to run the program without the `-t` option
  - Make a solution that allows you to wrap any program and get around the antivirus

Tasks *(continued)*

- Tips:
  - The antivirus has some useful command line parameters for debugging along the way
  - What happens when you run strings on the antivirus?