CS11 BINARY EXPLOITATION – LECTURE 1

Introduction/Reverse-Engineering basics

Course Overview

- Goal of the course is to teach you how to reverse engineer, identify bugs in, and exploit computer programs.
- Scope of this course will be limited to vulnerabilities usable for local privilege escalation – there are huge classes of bugs we won’t be touching.
  - Still, skills should be directly applicable for those interested! ☺
- CS24 should be considered a ***hard prerequisite!***
  - This class is going to require a lot of looking at and understanding assembly output by (potentially optimized) compilers.
  - Being unfamiliar or uncomfortable with x86 assembly or C will make this course very difficult.

Administrivia

- Six problem sets (five required).
  - One due every week - I have more material for those who enjoy it.
  - I tried to get IMSS to let me host VMs….ended up using AWS.
- Sets will be typically structured as “capture-the-flag” type problems, where your goal is to identify a bug, and find program input that allows you to recover a password.
  - Some minor exceptions, particularly in the first set.
  - First set will be short and reverse-engineering oriented – reverse a routine that decrypts a hardcoded password.
- Course will be focused on x86-64 assembly (and libraries, like libc).
- I (Bobby) will be holding office hours to present these lecture notes + help debug.

Roadmap

- Goal of the course is to teach you how to exploit bugs in programs.
- The first week will deal with reverse engineering - Figuring out what unknown assembly code does.
- After that, we'll talk about buffer overflows, out-of-bounds indexing, and memory corruption.
- Following that, we'll talk about **shellcode**.
- We'll then talk about C++ concepts – vtables and function pointers – and mitigations (ASLR/DEP).
- We'll finish up by talking about **ROP** (Return-Oriented Programming)

Why do people hack programs?

- Bugs are common, because low-level languages like C and C++ are difficult to write safe code in.
  - I’m sure most of you have encountered segfaults in your programs – and segfaults are usually exploitable.
- Bugs like segfaults allow a malicious attacker to gain **arbitrary code execution** on the device!
  - Install malware
  - Steal private/confidential data
  - Gain highly privileged information
  - Do anything a program you might run on your machine might do.

Why should you learn to exploit bugs?

- Defending against the types of attacks in the previous slide is important!
  - Nobody wants infrastructure-critical software, like that in self-driving cars to be exploitable.
- Reverse engineering proprietary software is fun!
  - Problems are difficult, and solving them is intensely rewarding.
- Understanding how common programming mistakes/bugs can be abused allows one to deeply understand how programs work in practice.
- Getting good at identifying bugs will help you write safer software of your own.
- Companies are often willing to pay those who disclose vulnerabilities through bug bounty programs, and the like.
  - Relatively niche skill, and there’s huge market for it.

Terminology

- **Binaries**
  - A compiled executable file (a program) containing machine code that can be run on some CPU. For example, an ELF, or a .o file.
- **Bugs/Vulnerabilities**
  - Unintended (abusable) behavior in a program.
- **Exploits**
  - Inputs or data (or tools that provide these) that, when applied to a binary, cause it to do something an attacker wants.
    - Typically, we'll be focusing on exploits that allow for control of the instruction pointer.

Review – how do computers work?

- CPUs are finite state machines, with some number of **registers**, connected to some **memory.**
- At each step, the CPU decodes an **instruction** in memory, which alters the state of its registers or other memory.
    - Dedicated register, the **instruction pointer** keeps track of where in memory the current instruction is.
    - This is also a huge simplification, but good enough for now.
- Typically, our goal when crafting an **exploit** will be to find a **vulnerability** that allows us to somehow get control of the instruction pointer.

Reverse Engineering

- **Reverse engineering** is the art of determining an unknown program’s purpose
- Before we can even get started trying to find or exploit bugs in code, we need to understand what code does.
- When trying to exploit binaries, you only very rarely have the luxury of symbols to tell you what functions are doing.
    - Need to be able to understand assembly language well enough to convert it into pseudcode (or C).

Reverse Engineering (II)

- Two “types” of RE:
- **“Static” Analysis**
  - Looking at a compiled binary file, and deducing its function from the assembly output/file contents alone.
- **“Dynamic” Analysis**
  - Determining a function by looking at it (often differentially) at runtime.
  - GDB is a good example of a dynamic analysis tool.
- We will be focusing primarily on static analysis techniques, but dynamic analysis is an extremely valuable tool!
  - Different tools for different use-cases.

Reverse Engineering - Process

- Consider the following assembly function – what does it do?:

```
my_function:
        enter $(8 * 2), $0
        mov $0, %rdx
        mov $0xCA, %rcx
        label1:
        movb %rdx(%rdi), %rax
        imul %rcx, %rax
        and  $0xFF, %rax
        movb %rax, %rdx(%rdi)
        inc %rdx
        cmp %rdx, %rsi
        jne label1
        leave
        ret
```

Reverse Engineering – Process

- Start by identifying trivial parts of a function – epilogue, prologue, etc.

```
my_function:
         enter $(8 * 2), $0                   ; Function Prologue
         mov $0, %rdx
         mov $0xCA, %rcx
         label1:
         movb %rdx(%rdi), %rax
        imul %rcx, %rax
        and  $0xFF, %rax
         movb %rax, %rdx(%rdi)
        inc %rdx
        cmp %rdx, %rsi
         jne label1
         leave                                ; Function Epilogue
         ret
```

Reverse Engineering – Process

- Identify data manipulations!

```
my_function:
        enter $(8 * 2), $0              ; Function Prologue
        mov $0, %rdx                    ; rdx = 0
        mov $0xCA, %rcx                 ; rcx = 0xCA
        label1:
        movb %rdx(%rdi), %rax           ; rax = rdi[rdx]
       imul %rcx, %rax                  ; rax *= rcx
       and  $0xFF, %rax                 ; rax &= 0xFF
        movb %rax, %rdx(%rdi)           ; rdi[rdx] = rax
       inc %rdx                         ; rdx++
       cmp %rdx, %rsi
        jne label1
        leave                           ; Function Epilogue
        ret
```

Reverse Engineering – Process

- Label loops as best as you can…

```
my_function:
        enter $(8 * 2), $0                 ; Function Prologue
        mov $0, %rdx                       ; rdx = 0
        mov $0xCA, %rcx                    ; rcx = 0xCA
        label1:                            ; while (rdx != rsi) {
        movb %rdx(%rdi), %rax              ;         rax = rdi[rdx]
        imul %rcx, %rax                    ;         rax *= rcx
        and  $0xFF, %rax                   ;         rax &= 0xFF
        movb %rax, %rdx(%rdi)              ; rdi[rdx] = rax
        inc %rdx                           ;         rdx++
        cmp %rdx, %rsi                     ;
        jne label1                         ; }
        leave                              ; Function Epilogue
        ret
```

Reverse Engineering – Process

- Convert to pseudocode and simplify.

```
my_function(data, length) {
        for (i = 0; i < length; i++) {
                data[i] = (data[i] * 0xCA) & 0xFF;
        }
}
```

Reverse Engineering – Process

- Reverse engineering can be difficult at first.
  - Luckily, it really does get easier the more you do it!
- Luckily, there are a lot of tools to aid in the process…

Tools

- strings
    - Prints out a list of ascii strings inside the program.
        - When reverse engineering a program, running `strings` on it should almost always be an early step
    - Programs oftentimes leave in debugging strings, and information intended to be output to the user is immensely helpful.

Tools

- xxd
  - Hex editor/viewer, allows you to look at the bytes making up a binary.
  - Much more useful than you’d think! Hex editors allow you to visualize data (not machine code) in a way that becomes secondhand to reason about with time.

Tools – Dealing with executables

- file
  - Can print out basic file format information for many, many unknown files.
    - Fails on custom formats, but still a valuable tool.
- readelf
  - Visualize a linux executable, exactly what it sounds like.

Tools - Disassembly

- objdump -d
  - Requires binutils to install.
  - Outputs the assembly instructions that makeup a program (to a text file)!
  - One of the easiest ways to get at the code that makes up a program.
    - Combined with a hex editor, you can patch binaries to edit their behavior!

Tools – Disassembly (II)

- IDA Pro
  - “Interactive Disassembler”
  - Pros:
    - Probably the best tool available on the market.
    - Extremely powerful, easy-to-use UI, allows for renaming variables, commenting, etc.
    - Visualizes programs as flow-charts (toggleable with space)
    - Also (optionally) comes with a de-compiler “hex-rays”, which automates much of the assembly->pseudocode process
      - Would recommend not using hex-rays for this class, as it may make it a bit too easy ☺
  - Cons:
    - Acquiring a new-ish copy is difficult to do legally.
      - A) It’s expensive – extremely so!
      - B) They don’t like selling to individuals because they’re worried they’ll buy once and then crack the program…
  - Luckily, there is a free version!
  - It’s fantastic tool…I can help you learn the shortcuts if you use it.

Tools – Disassembly (III)

- IDA Alternatives:
- radare2
  - https://github.com/radare/radare2
  - Open-source!
  - I'm not very familiar with it, but I've heard very good things about it.
- hopper
  - https://www.hopperapp.com/
  - Pretty good tool – I've used it and like it, but it's OS X only last I checked…
- binary ninja
  - https://binary.ninja/

Suggested Books/Resources

- Hacking: The Art of Exploitation
- Practical Reverse Engineering
- Exploiting Software: How to Break Code

- None are required, but may be helpful/interesting.

- https://microcorruption.com/login
  - Online “war game”, where you exploit increasingly more difficult bugs to “unlock” a virtual lock that you’re unlocking.
  - Uses a real instruction set (MSP430)
  - Incredible interface, I highly recommend it.
    - Early design for this class simply played these problems... ☺

Set 1 Overview

- The first set is pretty short.
  - (Sets 2 onwards will be longer, and will require using a VM)
  - I’ll get a VM image up pretty soon to use, just trying to figure out where to host it at the moment…
- Very basic reverse-engineering, nothing too rough.
  - Compile a program, run objdump on it, label the output.
  - Simple password tool – reverse a compiled binary
    - Validates user input against a hardcoded, “encrypted” password.
    - Given a compiled binary, figure out the correct, valid password.

Next Week

- Jumping into actual exploitation!
- “What is a buffer overflow, and how can it be exploited?”
    - What is the difference between overflowing on the heap versus on the stack?
- “What happens when programs access memory out-of-bounds?”