# Introduction to Reverse Engineering

Andrew Haberlandt

Feburary 9, 2021

- **Don't forget to start recording**
- Slides are on https://wiki.osucyber.club
- Some content adapted from: Rowan Hart (Purdue), Stephen Tong (Georgia Tech)

# Announcements

- Shirts and Stickers are In! (while supplies last)
  - Get 500 points in the Bootcamp CTF to get a sticker
  - Get 2000 points in the Bootcamp CTF to get a T-shirt
- Our CTF team got 40th out of 1000+ teams. Watch for announcements about future CTFs, and be sure to join us!
- **Next Week:** Intro to Binary Exploitation (a.k.a. pwn)


# Overview

# What is reverse engineering?

- **"...the process by which a man-made object is deconstructed to reveal its designs, architecture, code, or to extract knowledge from the object"**
- tl;dr Figure out how something works
  - Often with limited or no documentation
  - With a required level of understanding that varies depending on the task
- Examples
  - Figure out how a co-workers code works after they got fired
  - Figure out how a game communicates in a multiplayer session... for 'research' of course
  - Figure out how a streaming service plays audio/video... (note: piracy bad)

# About computers

## CSE 3421 in 3 minutes

- A computer understands a defined set of 'instructions', which each perform some small operation (arithmetic, data movement, I/O, control flow)
  - x86 vs. ARM vs. [...]
- Programmers should assume instructions run **'serially'** (one at a time, in sequence)
- Programs can interact with memory as a 'sea of byte-addressable bits'

| | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory | 47 | 6F | 20 | 42 | 75 | 63 | 6B | 72 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

# About computers

## CSE 3421 in 3 minutes

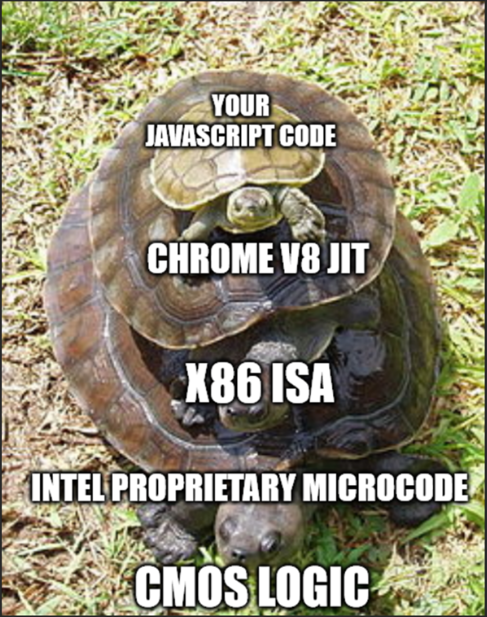
- "Move the integer '3' into register 2"
- "Get me 4 bytes at address 1012, store them in register 1"
- "Add register 1 to register 2, put the result in register 1"
- "Get me 1 byte using address in register 1, put the result in register 1
- "Move the data from register 1 into register 2"
- "If the value in register 1 is nonzero, jump to code at address 0x1337"

| | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory | 47 | 6F | 20 | 42 | 75 | 63 | 6B | 72 | 00 | 00 | 00 | 00 | E8 | 03 | 00 | 00 |

# Machine code is everything

or, everything is machine code

- **Native applications (compiled code):** Compile to machine code, distribute to users, then they run it
- **JIT-ed languages:** You write Javascript, the browser 'just-in-time' translates your Javascript into machine code, and then runs it.
- **Interpreted languages:** An interpreter (which is a native application) directly runs the source code, line by line, implementing all the behaviors of the language. e.g. CPython (the most common Python implementation)
- **Often, a combination of the above...** The line is blurry.


YOUR
JAVASCRIPT CODE
CHROME V8 JIT
X86 ISA
INTEL PROPRIETARY MICROCODE
CMOS LOGIC

# What does a reversing problem look like?

- **Native applications (compiled code):** You get machine code (usually in the form of an executable or shared library). You can use a **disassembler** to look at the instructions. A **decompiler** attempts to create C-like code from the machine code.
- **Java applications:** You'll usually get a .jar (or .apk if Android) containing 'Java bytecode', which can usually be decompiled back to nearly-runnable but hard-to-read Java
- **Web applications:** You can always view Javascript (+ HTML + CSS) in the browser (see the web talk). But minification/obfuscation tools can make it hard to understand.

# Demo 1

# What's usually missing in a reverse engineering problem?

- No useful variable names
- No documentation
- Obfuscated control flow
- But... we usually still have strings!

# Minified JS Example

**Pretty-Printed from Carmen**

```
function yn() {
  const e = Ba.t("This assignment is dropped and will not b
  const t = s()("#only_consider_graded_assignments").attr("
  const a = t ? "current" : "final"
  const n = ENV.group_weighting_scheme
  const o = !ENV.exclude_total
  const r = _n()
  s()(".dropped").attr("aria-label", "")
  s()(".dropped").attr("title", "")
  s()(".student_assignment").find(".points_possible").attr(
  l.a.forEach(r.assignmentGroups, e=>{
      l.a.forEach(e[a].submissions, e=>{
          s()("#submission_" + e.submission.assignment_id).
      })
  })
```

# Techniques for Reversing

## Static Analysis

- Stare at what you have, make sense of it piece-by-piece
- Use static analysis tools (decompilers, libraries such as `angr`, ...)

# Techniques for Reversing

## Dynamic Analysis

- Run it and see what happens; Modify inputs to see how behavior changes
- Debuggers, and other runtime inspection tools (Run it, pause it, poke around)
  - **Web:** Set breakpoints in browser developer tools, look at the argument values instead of staring at 'a', 'b', and 'c'.
  - **Native:** Use a debugger such as GDB (linux), windbg (windows); set breakpoints, inspect registers and memory
  - **Java / Android:** Frida (https://frida.re), and other runtime instrumentation tools
- Observe other program effects: Network activity (Wireshark / tcpdump), system calls (strace), ...

# Goodbye garbage collection

... when memory is just a sea of bytes

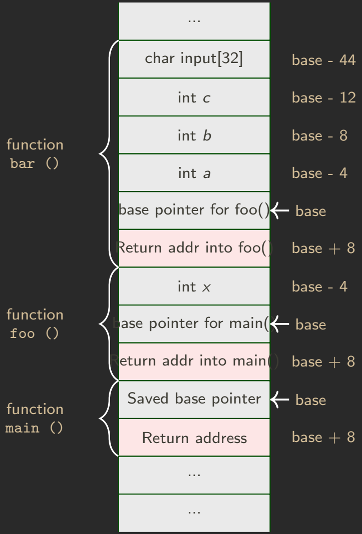
- **Pointers:** One register or memory location can have the address of another

| | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory | 47 | 6F | 20 | 42 | 75 | 63 | 6B | 72 | 00 | 00 | 00 | 00 | E8 | 03 | 00 | 00 |
| | "Go Bucks", null-terminated | | | | | | | | | | | | Addr (0x3E8) (Little endian) | | | |

- Interpretation of values in memory depends on **endianness** (little: 0x000003E8, big 0xE8030000)
- 0x3E8 is referencing the first character in the string (0x3E8 = 1000)

# The Stack

- **The stack grows toward LOW addresses**
- **Base pointer:** Holds the highest address in the current stack frame
- **Stack pointer:** Holds the current 'top' of the stack (lowest address currently in use)



# x86 Assembly Introduction

- tl;dr look up an instruction when you don't know what it does: https://www.felixcloutier.com/x86/
- **push/pop** - put things on stack / take things off stack
- **mov** - move data between registers and registers, or registers and memory
- **call, jmp** - call a function, jump to code at another address
- **add/sub/mul/xor/shl/shr/ and many more** - arithmetic, bitwise operations
- **conditions and control flow: cmp, test, jle, jge, jne, je** - comparisons, conditional jumps
- **Size suffixes:** movq = 8 bytes, movl = 4 bytes, movw = 2 bytes, movb = 1 byte
- **Intel vs. ATT syntax** - Ghidra uses Intel-ish
  - movq dst, src
  - movq %rax, %rbx - moves from rbx into rax
  - movq [%rax], %rbx - moves from rbx into memory at the address in rax

# C Intro

- Assembly is very verbose and hard to understand. C is easier.

```c
#include <stdio.h>
void main(void) {
  int x = 5;
  int y = 42;

  // Get the memory address of a variable
  int *x_ptr = &x;

  // Do address arithmetic
  int *ptr2 = x_ptr + 1; // adds sizeof(int) to x_ptr
  char *ptr2c = (char*)x_ptr + 4; // points to the same location

  // Get memory at an address ('dereference')
  // Because ptr2 is int*, we interpret the bytes as an int
  int mem = *ptr2;
  printf("ptr2: %p ptr2c: %p value: %d\n", ptr2, ptr2c, mem);
}
```

# Demo 2

# Reversing with GDB

- **Set breakpoints:** `b *0x1234`
- **Single-step through a program:** `step/next`
- **Change the value of a register:** `set %rax = 1234`
- **Display memory:** `x/32x %rsp` prints 32 bytes in hex, `x/s` prints as a string
- **... and more:** Google "gdb quick reference"

# Demo 3

# Tools

- Native code
  - Static: Ghidra, objdump; Python libraries: angr, pwntools; there may be plugins which help reversing native code compiled from C++ or other languages
  - Dynamic: GDB (linux), windbg (Windows), strace, frida; GEF for GDB
- Javascript
  - Chrome Developer Tools, ...
- Java / Android
  - Static: There are many decompilers. The one I demoed is jadx-gui.
  - Dynamic: Frida, ???

# Recommended Challenges

- **dump**: 20pt
- **java.lang.String**: 50pt
- **SoftwareShop**: 75pt
- **ransomware**: 180pt
- Also, any challenge 25 pt or less should be solvable before we talk about that category

# Next week

- **Highly Recommended:** Prepare a Linux x86 VM before next week
- https://wiki.osucyber.club/Bootcamp-CTF/Getting-Started/Environment
- I will be releasing a few more x86 reversing challenges this week