Customising Assembly

COMP3703 Software Security

Slides prepared by H. Gunadi and A. Tiu. Based on Chapter 4 & 8 of Andriesse's "Practical Binary Analysis", No Starch Press, 2019.

Outline

- Motivations
- Loading binary
  - Manual loading
  - Using libbfd
- Introduction to Capstone
  - Linear disassembly
  - Recursive disassembly
  - ROP gadget scanner

Why custom disassembly: Obfuscated code

- Most disassemblers output only a single disassembly listing
  - Assumption: each byte is mapped to at most one instruction, each instruction is contained in a single basic block, and each basic block is part of a single function.
  - Disassemblers typically assume that chunks of code don't overlap with each other.
- Instructions can overlap, breaking this assumption.
- Works in x86 because the ISA is dense, and the instructions have variable lengths.

Why custom disassembly: Overlapping code

```
$ objdump -M intel --start-address=0x4005f6 -d overlapping_bb
   4005f6: push   rbp
   4005f7: mov    rbp,rsp
   4005fa: mov    DWORD PTR [rbp-0x14],edi   ; load i
   4005fd: mov    DWORD PTR [rbp-0x4],0x0    ; j = 0
   400604: mov    eax,DWORD PTR [rbp-0x14]   ; eax = i
   400607: cmp    eax,0x0                    ; cmp i to 0
   40060a: jne    400612 <overlapping+0x1c>  ; if i != 0, goto 0x400612
   400610: xor    eax,0x4                    ; eax = 4 (0 xor 4)
   400613: add    al,0x90                    ; eax = 148 (4 + 144)
   400615: mov    DWORD PTR [rbp-0x4],eax    ; j = eax
   400618: mov    eax,DWORD PTR [rbp-0x4]    ; return j
   40061b: pop    rbp
   40061c: ret
```



```
$ objdump -M intel --start-address=0x400612 -d overlapping_bb
   400612:  add  al,0x4                     ; eax = i + 4
   400614:  nop
   400615:  mov  DWORD PTR [rbp-0x4],eax    ; j = eax
   400618:  mov  eax,DWORD PTR [rbp-0x4]    ; return j
   40061b:  pop  rbp
   40061c:  ret
```

Why custom disassembly

- Doing something general disassemblers aren't designed for.
- Omitting bogus code path.
- Creating hybrid-disassemblers.
- Cost and efficiency reason.

Loading binary

- Before we begin implementing a disassembler (using Capstone), we need to load the binary
- Need to load relevant information to start using Capstone.
  - At the very least, we need the information on the .text section, the size, and the assigned virtual address.
- We may also need other information such as what are the function addresses, etc to guide our custom disassembler.

Manual loading of binary

- Point Capstone to the right part of the program and load it.
  - E.g., find the size, virtual address, and load the .text section manually, and start capstone.
- However, things can get tedious quickly.
  - As we may need information from different parts o the binary.
  - Handling different binary format (ELF/PE) or other ISAs other than x86.
- There's already a library for loading binary (libbfd)– why reinventing the wheel?

Libbfd: a quick intro

- A common interface for reading / parsing all popular binary formats
- Compiled for a wide variety of architectures
  - Includes ELF and PE files for x86 and x86-64 machines.
- Used by many applications in the binutils suite
  - e.g., objdump, readelf, and gdb.
- Provides generic abstractions for all binary components:
  - headers describing the binary's target and properties,
  - lists of sections,
  - symbol tables,
  - etc.

A simple interface for libbfd

- For this course, we need to perform a few simple tasks.
- We'll use a wrapper of libbfd: the loader library.
  - Chapter 4 of Practical Binary Analysis.
- The loader library revolves around 3 main classes: `Binary`, `Sections` and `Symbols`.
- Two important functions implemented: `loads` and `unloads`
  - The rest are accessing the information through the classes.
  - The actual libbfd API functions are those starting with `bfd_`.

The loader library – general workflow

- Initialise bfd
- Open the binary file & checking the file format
- Get necessary information about the executable header.
- Load Symbols
- Load Sections
- Organize the loaded information

pwnlib.elf: a python library for ELF

- An alternative to [redacted] a python library for ELF processing (pwn[redacted])
  - Pwnlib.elf is part of pwntools, a collection of python libraries for reverse engineering and exploitation developments.
  - It is based on another library called **elftools**.
- This can be useful for quick and simple analysis tasks – easy to set up and platform independent.
- We'll look at some simple examples of querying information about an ELF binary.
  - See https://docs.pwntools.com/en/stable/elf.html for more details

pwnlib.elf: a python library for ELF


```
# load dependenci
>>> from pwn impo

# load an ELF binary
>>> e = ELF('compilation_example')

# list all symbols
>>> e.symbols

# query the address of 'main' symbol
>>> e.symbols['main']
```

pwnlib.elf: functions

```
# list all functi
>>> e.functions

# show details of `main` function
>>> e.functions['main']

Function(name='main', address=0x401126, size=0x22,
elf=ELF('/home/binary/lectures/capstone/compilation_example'))
```

pwnlib.elf: sections

```
# list all sections (returns a list of Section objects)
>>> e.sections

# get a section by name (returns a Section object)
>>> s=e.get_section_by_name('.text')

# query the raw data of section (when loaded to memory)
>>> s.data()

# query the section header
>>> s.header
Container({'sh_name': 148, 'sh_type': 'SHT_PROGBITS', 'sh_flags': 6,
'sh_addr': 4198464, 'sh_offset': 4160, 'sh_size': 389, 'sh_link': 0,
'sh_info': 0, 'sh_addralign': 16, 'sh_entsize': 0})

# get the address of the section header
>>> s.header['sh_addr']
4198464
```

Introduction to Capstone

- Capstone is a di[illegible]ly framework designed to provide a simple, [illegible]ight API
- It transparently handles most popular instruction architectures, including x86/x86-64, ARM, and MIPS, among others.
- It has bindings for C/C++ and Python (plus other languages).
  - We'll look at both C/C++ and Python bindings
  - See http://www.capstone-engine.org for other supported languages.

Introduction to Capstone

- It runs on all popular platforms, including Windows, Linux, and macOS.
- It's also completely free and open source.
- Simple yet powerful.
  - recover virtually all relevant details of disassembled instructions, including instruction opcodes, mnemonics, class, registers read and written by the instruction, and more.
- Important information in /usr/include/capstone/capstone.h and x86.h (Since we are dealing with x86).

Capstone: simple examples

- Linearly disasse[illegible]tructions into a human-readable form, or instructi[illegible]nonics.
- Takes a buffer containing a block of code bytes as an input (.text)
  - outputs instructions disassembled from those bytes.
- 3 major parts:
  - Some initialization,
  - Call to cs_disasm API function, and
  - output-parsing code.

Initialising Capstone

- cs_open(): open properly configured Capstone instance.
  - In our case, set up to disassemble x86-64 code.
- CS_ARCH_X86: disassemble code for the x86 architecture.
- CS_MODE_64: disassemble 64-bit architecture.
- Will store the result in the the third argument.
  - a pointer to an object of type csh ("Capstone handle").
- This handle is needed to invoke any of the other Capstone API functions.
- CS_ERR_OK: successful cs_open().

Disassembling Code Buffer

cs_disasm(): main [illegible]sassemble the whole block of code.

- Takes the Capstone handle,
- Buffer containing the code,
- Size,
- Virtual Memory Address (VMA),
- Number of instructions to disassemble, and
- Output buffer (disassembled instructions).

CS_INSN Structure

- cs_disasm() buil
  array of disasse
  instructions in the process.
- The id values are unique only within architecture.
  - Useful for comparing instructions (more reliable than string comparison).
- The detail field contains more detailed information for advanced disassembly.

```
typedef struct cs_insn {
    unsigned int    id;
    uint64_t        address;
    uint16_t        size;
    uint8_t         bytes[16];
    char            mnemonic[32];
    char            op_str[160];
    cs_detail       *detail;
} cs_insn;
```

Setting up detailed disassembly mode

- Controlled through the OPT_DETAIL option.
- Provide more detailed information:
  - e.g., registers accessed, the type and value of its operands, the type of instruction (arithmetic, control flow, and so on), or the locations targeted by control flow instructions.
  - But it will make the disassembly process slower.
- This more detailed information can be used to guide the disassembly process.
  - Hence, usually it is paired with the iterative disassembly as opposed to batch disassembly.

Recursive disassembly

- Uses queue of entry points.
  - Starting with entry point and known functions.
  - And add branch targets as they are disassembled.
- Linearly disassemble each of the entry in the queue.
- Stops when the disassembler sees `hlt` instruction or unconditional branch instruction.
  - Those instructions do not have guaranteed fall-through instruction.

Recursive disassembly: iterative disassembly

- For real-time instruction parsing
- cs_disasm_iter(): disassemble one at a time.
  - false when there is no more instruction to disassemble.
- Keep the pointer to the bytes of code to disassemble.
  - Update the pointer after each call.
  - Akin to program counter.
- Also keep the bytes left to disassemble and the VMA.
- Faster and more memory efficient.

Recursive disassembly: parsing control flow instructions

- Uses the group information of an instruction.
  - No need to enumerate all of the jump instructions.
- Does not attempt resolve indirect control flow.
- Resolving control flow is architecture specific:
  - Because we need to see the operands, and their encodings are architecture specific.

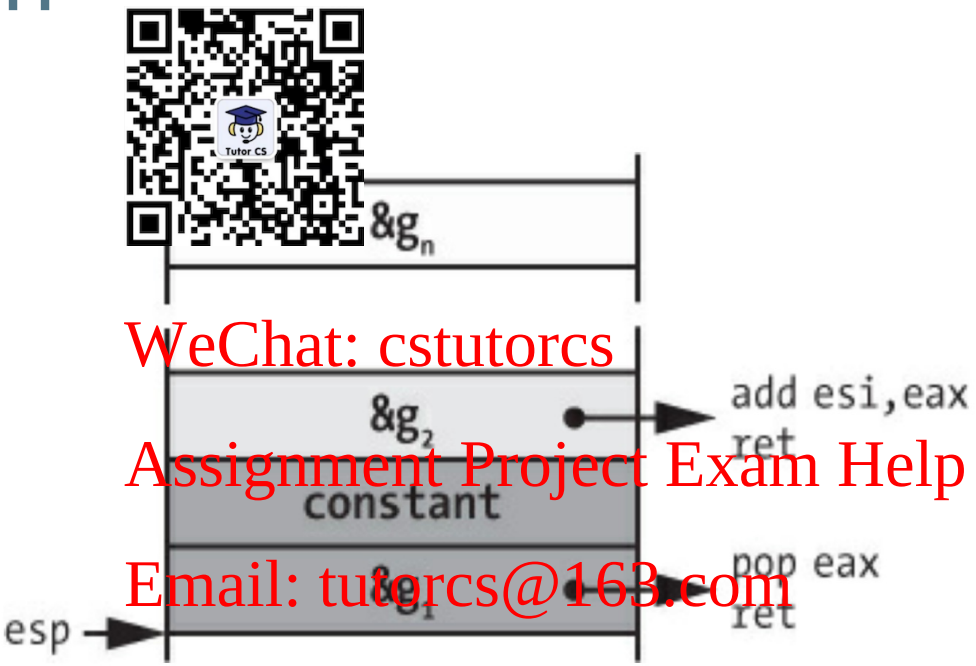
Return-Oriented Programming

- A technique to get around the stack smashing protection (eg, stack guard).
- Return-to-libc: instead of injecting code to stack, redirect control to sensitive functions (e.g., libc execve).
- ROP generalises return-to-libc to allow an attacker to chain together existing code sequences in the target program memory.
- These code sequences are called **gadgets**.
- We'll cover ROP in more detail later in the course – now we focus on the problem of finding gadgets.

ROP gadgets

- Each gadget ends in a return instruction and performs a basic operation (eg., addition or logical comparison).
- Gadgets can be combined to form a custom instruction, which is used to craft arbitrary functionality.
- An ROP program consists of a series gadgets, such that the return instruction terminating each gadget transfers control to the next gadget.
- To start an ROP program, execute an initial return to instruction to jump to the first gadget address.

ROP chain



An example of ROP chain. Gadget g1 loads a constant into eax, which is then added to esi by g2.

ROP: Finding Gadgets

- Limit to length 5.
- Both aligned and unaligned instructions.
- Naive: iterate over all possible starting byte.
- We can be smarter, start from ret instead.

Scanning for Roots and Mapping Gadgets

- We can search for the instruction in the code bytes.
- No need to run through the disassembly process.
- The resulting gadget can be stored for further processing or printed directly.
- Don't forget to map it to the address where the ROP starts.

Finding all gadgets at a given root

- Once we find the location of ret, we can search from the location -1, up to (-15).
- Each instruction can be at most 15 bytes long.
- Move on to the next possible address when:
- Hitting jump instruction, or
- Hits an instruction beyond ret, or
- If it is longer than the desired length (5), or
- Hitting invalid instruction.

The python bindings for Capstone

- Most Capstone APIs are available in its python bindings.
- Iterative disassembly, supported in C++ bindings, is not available in its python counterparts.
- For simple binaries, the python bindings may be more convenient, though probably less efficient.
- The overall code structures for disassembly are very similar to C++.
  - See the provided code on Wattle for details.

Summary

- We have covered the basics of capstone API to perform simple disassembly.
- We have also applied it to implement an ROP gadget scanner.
- We'll cover some more exercises in the lab, and examining ROP in more details.