MITLL CTF Tutorial

Binary Analysis and Exploitation

William Robertson

19 Oct 2013

Northeastern University

Welcome

- Today, we discuss the analysis and exploitation of (ARM) binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.

Welcome

- Today, we discuss the analysis and exploitation of (ARM) binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model.

Welcome

- Today, we discuss the analysis and exploitation of (ARM) binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model.
  2. Understand the structure of binaries.

Welcome

- Today, we discuss the analysis and exploitation of (ARM) binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model.
  2. Understand the structure of binaries.
  3. Learn static and dynamic techniques for analyzing binaries.

Welcome

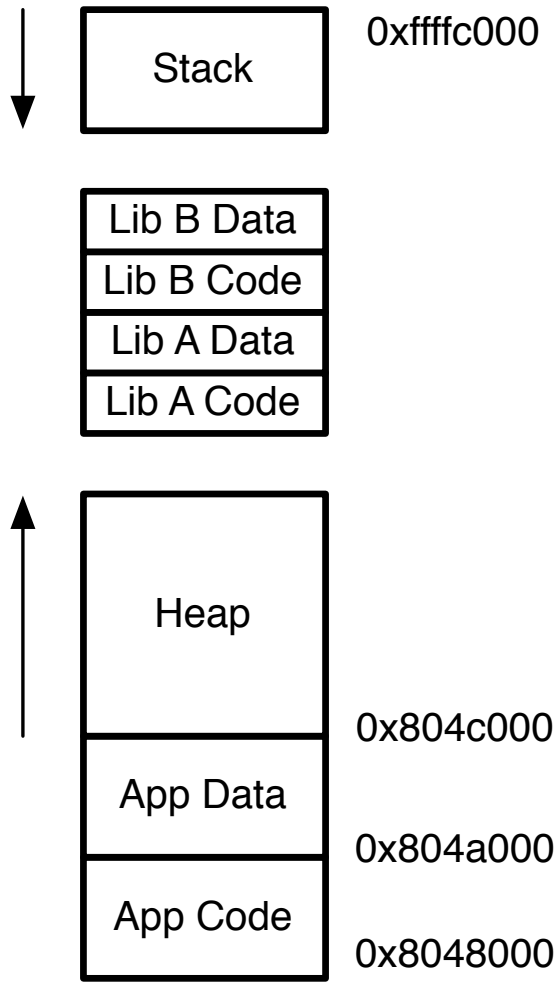
- Today, we discuss the analysis and exploitation of (ARM) binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model.
  2. Understand the structure of binaries.
  3. Learn static and dynamic techniques for analyzing binaries.
  4. Demonstrate exploiting a vulnerable program.

Process Execution

A *process* is a virtual address space and one (or more) threads of control.

- Memory.
- Stack.
  - Function activation records.
  - Local variables.
- CPU.
  - Function arguments (r0 – r3)
  - Local variables (r4 – r11)
  - Stack pointer (r13)
  - Link register (r14)
  - Program counter (r15)

Process Memory Layout



Executable Formats

- Binary programs consist of code, data, and metadata.
- Variety of formats:
  - PE32 (Windows)
  - ELF (UNIX)
  - COFF (UNIX)
  - a.out (UNIX)
- We will focus on Linux-based ELF binaries.
  - But, the main principles apply to other formats.

ELF

- **E**xecutable and **L**inkable **F**ormat.
- ELF header.
  - ELF magic, architecture, flags, entry point, etc.
- Program header.
  - Refers to *segments*.
  - Segments related to runtime process memory layout, i.e., code and data.
- Section header.
  - Refers to *sections*.
  - Linking and relocation data.
  - Debugging information.

Binary Analysis

- Given a binary, we want to learn something about it.
  - Understand its intended behavior and security policies.
  - Recover some sensitive data, hijack control flow to execute malicious code, ...
- Two main approaches.
  - Statically (disassembly and some automated analysis).
  - Dynamically (observe execution over concrete inputs).

Disassembly

- Disassembly recovers instructions from machine code in binary format.
- Useful for getting an idea of what the program does.
- Tools.
  - `objdump`
  - `IDA Pro` (expensive, but nice)

Disassembly

- Disassembly recovers instructions from machine code in binary format.
- Useful for getting an idea of what the program does.
- Tools.
  - `objdump`
  - `IDA Pro` (expensive, but nice)
- Today, we'll focus on `objdump`.

Stack Overflows

- Fundamental problem is that control flow information is stored inline with app data.
  - Low-level languages like C don't strictly enforce integrity of control data.
- There are a number of easy ways to corrupt this data.
  - For instance, by writing past the end of a stack-allocated buffer.
  - `strcpy`, `memcpy`, app-level loops.
- Overflows can allow untrusted users to control return address values.
  - What happens when a `ret` instruction is executed?
  - Return value overwrites are not the only possibility, of course.

Assuming Control

- By overflowing a local buffer, we can control the program counter and jump to code of our choosing.
- Options?

Assuming Control

- By overflowing a local buffer, we can control the program counter and jump to code of our choosing.
- Options?
    1. Inject a payload (e.g., shellcode).

Assuming Control

- By overflowing a local buffer, we can control the program counter and jump to code of our choosing.
- Options?
    1. Inject a payload (e.g., shellcode).
    2. Return into libc.

Assuming Control

- By overflowing a local buffer, we can control the program counter and jump to code of our choosing.
- Options?
    1. Inject a payload (e.g., shellcode).
    2. Return into libc.
    3. Return-oriented programming.

Assuming Control

- By overflowing a local buffer, we can control the program counter and jump to code of our choosing.
- Options?
    1. Inject a payload (e.g., shellcode).
    2. Return into libc.
    3. Return-oriented programming.
- For our demo, let's return into libc.

ret2libc

- ret2libc attacks are simple examples of *code reuse* attacks.
  - We want to jump to an existing function with arguments we specify.
- On x86, the goal is to overwrite the return address and set up function arguments on the stack.
- But, ARM binaries don't pass arguments on the stack!
  - Instead, arguments passed in r0 to r3.
  - Is there a way around this?

Gadgets

- In order to pass arguments, we need to find “gadgets” that load r0 – r3.
- We can chain gadgets by finding instruction sequences that end in a return – i.e., `pop pc`.
- Let’s see an example!

Conclusions

- We reviewed process execution and binary program structure.
- We learned simple static and dynamic techniques for analyzing binaries.
- We developed an end-to-end exploit for a basic stack overflow.

Next Steps

- This is just the tip of the iceberg!
- More attacks.
  - Heap overflows.
  - Format strings.
  - atexit, .ctor, .dtor, PLT/GOT overwrites.
- Defenses.
  - Stack, heap cookies.
  - Address space layout randomization (ASLR).
  - Non-executable memory.
  - Control flow integrity (CFI).
  - Obfuscation (packing, anti-debugging).

Next Steps

- This is just the tip of the iceberg!
- More attacks.
  - Heap overflows.
  - Format strings.
  - atexit, .ctor, .dtor, PLT/GOT overwrites.
- Defenses.
  - Stack, heap cookies.
  - Address space layout randomization (ASLR).
  - Non-executable memory.
  - Control flow integrity (CFI).
  - Obfuscation (packing, anti-debugging).
- Low-level exploitation is fun, and the skills are in demand.

Thanks for your attention!

Questions?

<wkr@ccs.neu.edu>