PWN College

Session 2

Atousa Ahsani

Main Reference: https://pwn.college/

Fundamentals

Computer Architecture

Assembly Code

Introduction to Binary Files

Linux Process Loading

Linux Process Execution

Assembly

- The only true programming language, as far as a CPU is concerned.

- Concepts:
  - Registers
  - Instructions
  - Memory

Memory (stack)

- The stack is a **hardware manifestation** of the **stack data structure**.
- The stack is simply an **area** in **RAM**.
  - There is no special hardware to store stack contents.
- Relevant registers (amd64): ***rsp***, ***rbp***
  - The ***esp/rsp*** register holds the address in memory where the bottom of the stack resides.
  - The ***ebp/rbp*** contains the address of the top of the current stack frame.
- A **stack frame** is essentially just the space used on the stack by a given function.
- Relevant instructions (amd64): ***push***, ***pop***

Memory (stack)

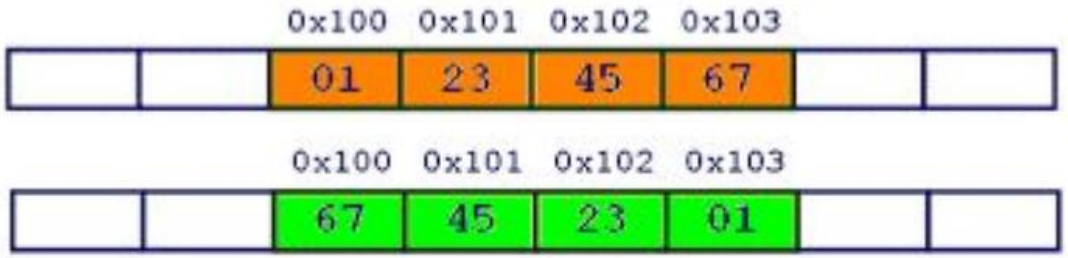
- When something is **pushed** to the stack, ***esp*** decrements by 8 (on 64-bit arch), and the value that was pushed is stored at that location in memory.
- when a **pop** instruction is executed, the value at ***esp*** is retrieved , and ***esp*** is then incremented by 8.
- *Note*: The stack "grows" **down** to lower memory addresses!
- The stack fulfils three main uses:
  1. Track the "**callstack**" of a program.
     - return values are "*pushed*" to the stack during a **call** and "*popped*" during a **ret**.
  2. Contain **local variables** of functions.
  3. Pass **function arguments** (always on x86, only for functions with "many" arguments on other architectures).

Memory (other mapped regions)

- Functionalities such as ***mmap*** and ***malloc*** can cause other regions to be mapped as well.
- ***mmap()***
  - A Unix system call that maps **files** or **devices** into **memory**.
- ***malloc()***
  - It is used to dynamically allocate a single large block of memory with the specified size.

Memory (endianess)

- On most modern systems, data is stored backwards, in **little endian**.



- Why?
  - Performance
  - Ease of addressing for different size.

Signedness: Two's Compliment

- How to differentiate between positive and negative numbers?

- One idea: **Signed Bit** (8-bit example):

```
b00000011 == 3
b10000011 == -3
```

drawback 1: `b00000000 == 0 == b10000000`

drawback 2: arithmetic operations have to be signedness-aware

```
(unsigned) b11111111 + 1 ==  255 + 1 ==   0  == b00000000
(signed)   b11111111 + 1 == -127 + 1 == -126 == b11111110
```

Signedness: Two's Compliment

- Clever (but crazy) approach: **Two's Complement**

```
b00000000 == 0
 0 - 1 == b11111111 == 0xff == -1
-1 - 1 == b11111110 == 0xfe == -2
```

advantage: arithmetic operations don't have to be sign-aware!

(unsigned) `b11111111 + 1 ==  255 + 1 == 0 == b00000000`

(signed) `b11111111 + 1 ==  -1 + 1 == 0 == b00000000`

disadvantage: you might go crazy

- The fundamental arithmetic operations of addition, subtraction, and multiplication are **identical** to those for unsigned binary numbers.

Calling Conventions

- **Callee** and **caller** functions must agree on argument passing.
- Linux x86:
  - push arguments (in **reverse order**),
  - then call (which pushes **return address**),
  - return value in ***eax***.
- Linux amd64:
  - ***rdi***, ***rsi***, ***rdx***, ***rcx***, ***r8***, ***r9***, return value in ***rax***.
  - Any remaining arguments are passed on the stack in **reverse order** so that they can be popped off the stack in order.
- Linux arm:
  - ***r0***, ***r1***, ***r2***, ***r3***, return value in ***r0***

Calling Conventions

- Registers are **shared** between functions, so calling conventions should agree on what registers are protected.
- When one function calls another, the former is the **caller** and the latter is the **callee**.
- The callee must **save** and **restore** any **preserved registers** that it wishes to use. The callee may change any of the **nonpreserved** registers.
- Hence, if the caller is holding active data in a **nonpreserved** register, the **caller** needs to save that **nonpreserved** register before making the function call and then needs to restore it afterward.
- For these reasons, **preserved registers** are also called **callee-save**, and **nonpreserved registers** are called **caller-save**.

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return `long double` arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

http://galaxy.agh.edu.pl/~amrozek/AK/abi.pdf

Other Resources

- Opcode listing:
  - http://ref.x86asm.net/coder64.html
- x86_64 architecture manual:
  - https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
- Rappel lets you explore the effects of instructions.
  - https://github.com/yrp604/rappel
  - Easily installable via https://github.com/zardus/ctf-tools

Fundamentals

Computer Architecture

Assembly Code

Introduction to Binary Files

Linux Process Loading

Linux Process Execution

What is an ELF?

- **E**xecutable and **L**inkable **F**ormat
- ELF file extension format is a common standard file extension used for **executable**, **object code**, **core dumps** and **shared libraries**.
- It was being chosen as the standard binary file format for **Unix** and **Unix-based** systems.
- **Magic number** of an ELF file:

```
➜  ~ xxd -l 16 a.out
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
```

  - A **constant value** used to identify a file format, protocol or error code.

What is an ELF?

- Contains the **program** and its **data**.
  - **Program/Segment Headers**: Describes how the program should be **loaded**.
  - **Section Headers**: Contains **metadata** describing program **components**.

ELF Program Headers

- **Program headers** specify information needed to prepare the program for execution. They define **segments** and their type.
- Each **segment** contains **information** that is needed for **run time** execution of the file.
- A **segment** is loaded into the memory when that file is executed.
- There are several different kinds of program header. The most important entry types are:
  - **INTERP**: defines the **library** (the name of the **program interpreter**) that should be used to load this ELF into memory.
  - **LOAD**: defines a **part of the file** that should be loaded into memory.

ELF Program Headers

- **readelf**:
  - Display information about the contents of ELF format files.

```
→  2- BinaryFiles readelf -a cat
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x10e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14888 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

ELF Program Headers

- Each segments contains different data, So the permissions are different.
- **Program headers** are the only source of information used when loading a file.

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000700 0x0000000000000700  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000335 0x0000000000000335  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x0000000000000160 0x0000000000000160  R      0x1000
  LOAD           0x0000000000002d98 0x0000000000003d98 0x0000000000003d98
                 0x0000000000000278 0x0000000000000280  RW     0x1000
  DYNAMIC        0x0000000000002da8 0x0000000000003da8 0x0000000000003da8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000000358 0x0000000000000358
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x0000000000002014 0x0000000000002014 0x0000000000002014
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002d98 0x0000000000003d98 0x0000000000003d98
                 0x0000000000000268 0x0000000000000268  R      0x1
```

ELF Section Headers

- A different view of the ELF with **useful information** for introspection, debugging, etc.
- Important sections:
  - **.text**: It is a code section that contains program **code instructions**.
  - **.plt**: It stands for **Procedure Linkage Table**. It is used to call **external procedures/functions** whose address isn't known in the time of linking, and is left to be resolved by the dynamic linker at run time.
  - **.got**: It stands for **Global Offsets Table** and is similarly used to resolve addresses

ELF Section Headers

- Important sections (cont'd):
  - **.data**: used for **pre-initialized global writable** data (such as global **arrays** with initial values)
  - **.rodata**: used for **global read-only** data (such as **string constants**)
  - **.bss**: used for **uninitialized global writable** data (such as global arrays without initial values)

- **Section headers** are not a necessary part of the ELF.
  - Only segments (defined via **program headers**) are needed for **loading** and **operation**! Section headers are just **metadata**.