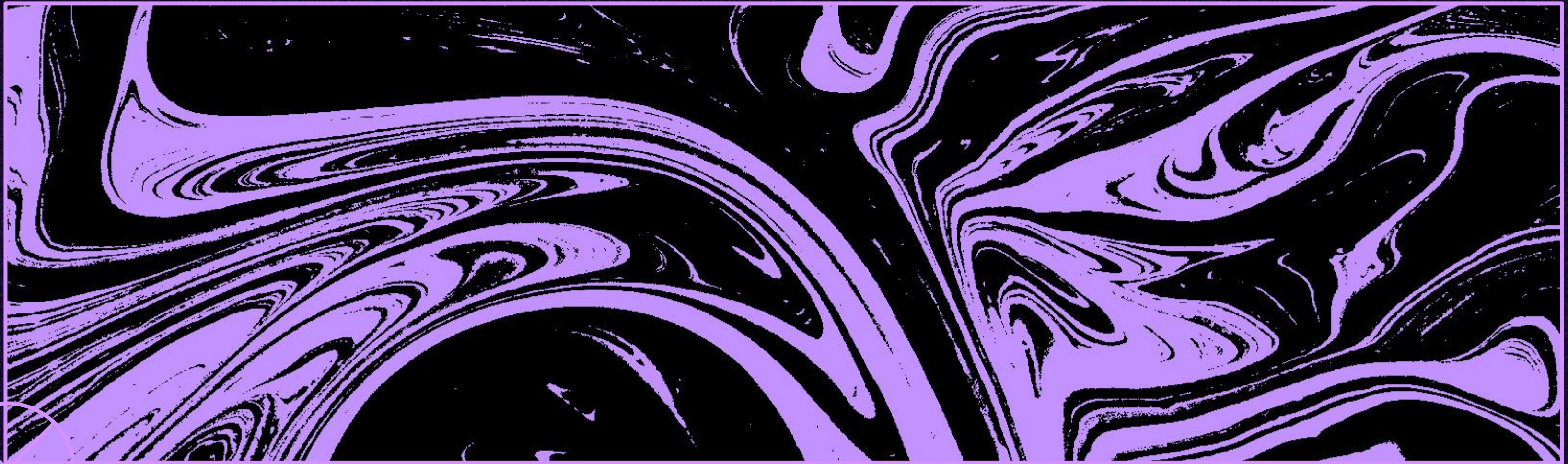
Reverse Engineering

For Beginners

x86_64



x86_64

Linux x86_64

TABLE OF CONTENTS

Duration: 2 Hours

Intro

Intro

01.

Intro

#Whoami

Samandeep Singh

- Organizer - BSides Singapore
- 9+ years in Infosec
- Interested in vulnerability research (OpenSource)
- Learning solidity smart contract security
- Tweet me @samanl33t

WHAT - WHY - HOW ? RE

At the core: *To dismantle an object to see how it works.*

For software: (*roughly*) *To disassemble the compiled code to understand it's logic with the intention to circumvent or replicate it.*

WHY?

- Military use cases
- Vulnerability Research
- Malware Analysis
- CTFs and curiosity
- Historically:
  - Cracks
  - Keygens

In the context of Linux ELF binaries:

- Programming/ Development process:

  Code (mainly C/C++) > Compile > Assemble

  Generates an ELF Binary.

- Reverse Engineering process:

Disassemble (ELF Binary) > Decompile > Lots and lots of Reading > 5 stages of grief > Voila!!

Theory

02.

X86_64 Basics (Theory)

Theory

x86_64 ASSEMBLY

- The lowest level of programming language for CPUs
- 64-bit version of x86 instruction set
- Consists of
  - Registers (rbp, rsp etc.)
  - Instructions (mov, cmp, jmp etc.)
  - Memory (stack, program data, heap etc.)

REGISTERS

- Special kind of super fast memory to work with instructions.
- 64-bit/ 8 byte wide
- Can be accessed partially
  - For e.g: Lower WORD (2 bytes), DWORD (4 bytes)
- Must know types:
  - General Purpose Registers (16)
  - Pointer Registers (1)
  - RFlags (3)

General Purpose Registers

64-bit	32-bit	16-bit	8H	8L	Description
rax	eax	ax	ah	al	Accumulator Register
rbx	ebx	bx	bh	bl	Base Register
rcx	ecx	cx	ch	cl	Counter Register
rdx	edx	dx	dh	dl	Data Register
rsp	esp	sp		spl	Stack Pointer
rbp	ebp	bp		bpl	Base Pointer
rsi	esi	si		sil	Source Index
rdi	edi	di		dil	Destination index

General Purpose Registers

64-bit	32-bit	16-bit	8H	8L	Description
r8	r8D	r8W		r8B	General Purpose
r9	r9D	r9W		r9B	General Purpose
r10	r10D	r10W		r10B	General Purpose
r11	r11D	r11W		r11B	General Purpose
r12	r12D	r12W		r12B	General Purpose
r13	r13D	r13W		r13B	General Purpose
r14	r14D	r14W		r14B	General Purpose
r15	r15D	r15W		r15B	General Purpose

Pointer Registers & RFlags

64-bit	32-bit	16-bit	8H	8L	Description
rip	eip	ip			Instruction Pointer

RFlag	Description
CF	Carry Flag
SF	Sign FLag
ZF	Zero Flag

x86_64 ASSEMBLY

INSTRUCTIONS

	OPCODE	OPERAND	OPERAND
	Operation/What to do	What to do with/on	
Eg:	mov	rax	rsp

x86_64 ASSEMBLY

INSTRUCTIONS

Different Types:

- Data Manipulation:
    - mov rsp,rax
    - inc [rax]
    - Arithmetic ops.: add rax, r9 / sub rax,r9 / mul rax, r9
- Control Flow & Conditions:
    - Jumps: je, jne, jg,jge, jle etc.
    - cmp, test etc.
- System Calls:
    - To interact with the OS.
    - System call (syscall) numbers are used (rax)
    - Eg: Read, write, open etc.

MEMORY

Stack

- Last-in First-Out (LIFO)
- Grows toward lower memory addresses/backwards.
- Used to store data temporarily during program execution. Eg:
    - Storing and using function return values (push & pop instructions)
    - Stores function local variables
    - Sometimes, used to pass function arguments.

Heap/Others

- Dynamically mapped regions in the memory:
    - Eg. using malloc(), mmap() etc.

Stack Layout

```
                          0x00000000 (Lower Memory address)
                          ___________
                          |         |
                     ESP  |         |
PUSH 0xEE                 |  0xEE   |
                          |  0xCC   |
                          |  0xAA   |
                          ---------------
                          0xFFFFFFFF (Higher Memory Address)
```

- PUSH: Pushes 8 bytes to the top of stack
- POP: Pops 8 bytes from the top of stack
- RSP: Stack pointer gets updated based on the instructions. (+/- 8 bytes)

x86_64 ASSEMBLY

Endianess

- The order in which data bytes are arranged into larger numerical values
- Intel x86_64 CPUs follow Little Endian format
    - i.e the bytes are ordered from least significant bit.

	Low address							High address
Address	0	1	2	3	4	5	6	7
Little-endian	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Big-endian	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Memory content	0x11	0x22	0x33	0x44	0x55	0x66	0x77	0x88

64 bit value on Little-endian	64 bit value on Big-endian
0x8877665544332211	0x1122334455667788

Ref: https://www.nekosecurity.com/assets/images/assembly/endianness.png

x86_64 Calling Conventions

How a function call is made in Assembly?

- Passing arguments:
  - Using registers: rdi, rsi, rdx, rcx, r8, r9
  - Uses stack: When there are many arguments.
- Return values:
  - returned in rax
- Callee-saved registered: rbx, rbp, r12, r13, r14, r15

```
int sum (a,b){
Int c;
c = a + b;
return c;
}


int result;
result = sum(1,5);
// rdi = 1;
// rsi = 5
// rax = 6 = result
```

x86_64 Linux Syscalls

- For the program to communicate with outside world (OS)
- Approx. 330 syscalls in linux
- Calling convention:
    - Syscall number passed to rax
    - Pass arguments in registers (rdi, rsi,rdx etc.)
    - call ‘syscall’ instruction.
    - Eg: system call: exit(1);

```
exit:
    mov rax, 0x3C
    mov rdi, 1
    syscall
```

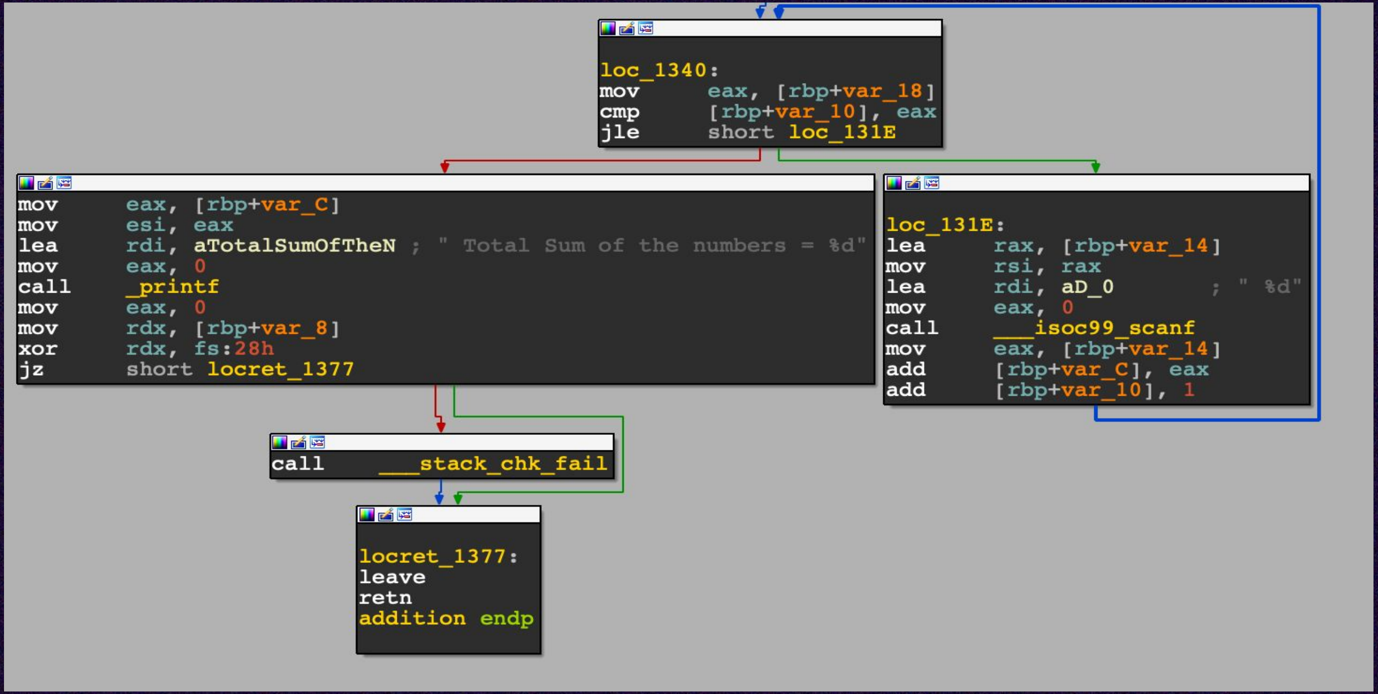
Syscall Table: https://x64.syscall.sh/

x86_64 Functions & Frames

Simply put:

- Every program consists of multiple functions (within different modules)
  - Modules can be internal or external
- Each function is a combination of multiple blocks of instructions
  - Instructions => Assembly instructions
- The function usually has a specific goal. Eg:
  - read/write/update data
  - call other functions etc.
- The combination of blocks can be represented as Control Flow Graph.
  - Where each instruction is executed one after another
  - And the flow is defined by different conditions

Control Flow Graph



x86_64 Functions & Frames

Before a function is called:

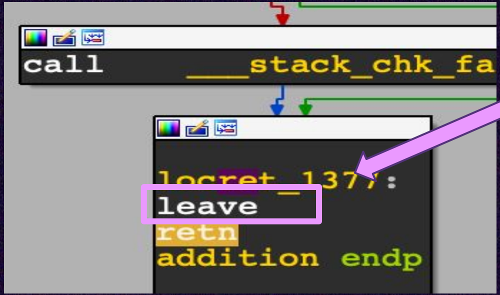
- The address to return to is pushed on the stack.
- Function arguments are moved to registers (sometimes on stack if required)

```
loc_1229:
mov     eax, 0
call    addition
jmp     short loc_1281
```

x86_64 Functions & Frames

When a function is called:

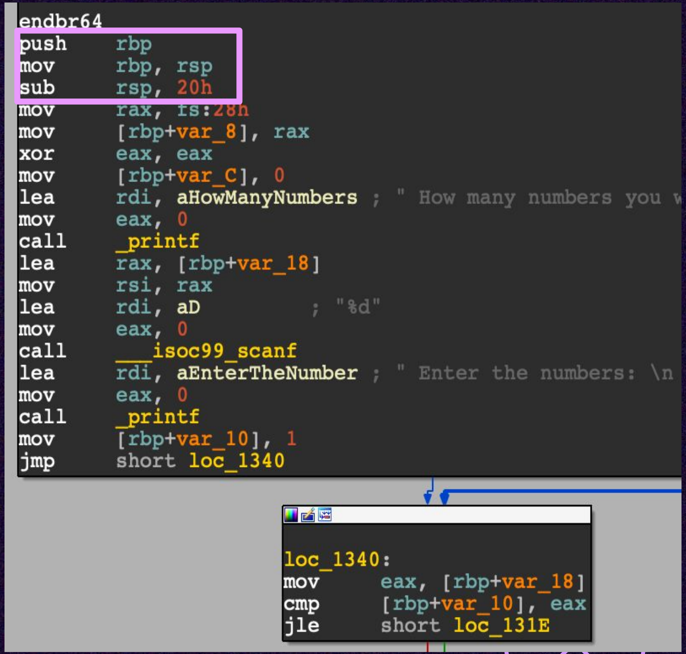
- Sets the stack frame - Prologue
- Executes the Instructions inside
- Tears down the stack frame - Epilogue



leave :

mov rsp,rbp

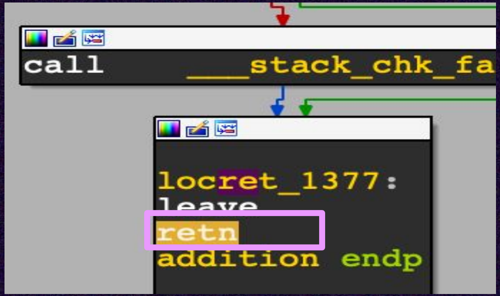
pop rbp



x86_64 Functions & Frames

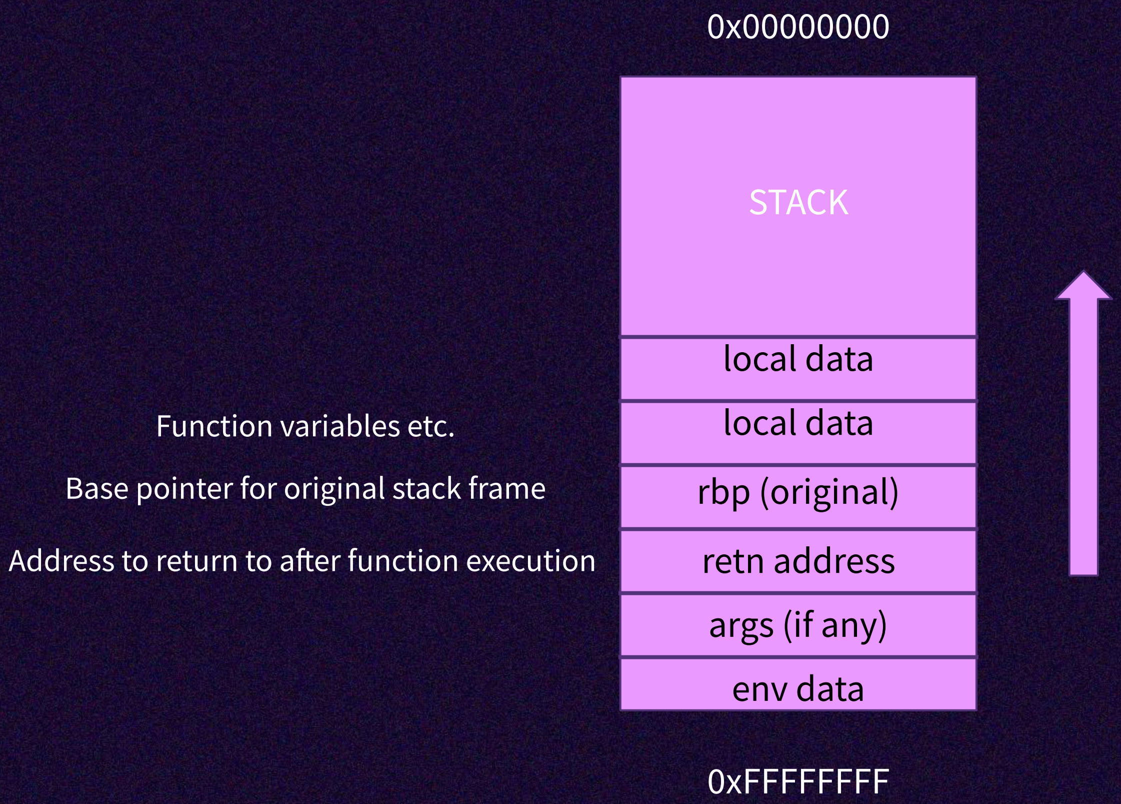
When all instructions in the function are executed

- A return instruction is called, which:
  - Pops the previously stored return address from stack



Marketing plan

Function Stack Frame



Lab

Lab

03.

Reverse Engineering - Basics (Lab)

RE Basics Lab Intro/Setup

Lab Instructions

- Clone the repository:
  - https://github.com/samanL33T/x86_64_RE_For_Beginners_STANDCON_2022.git
- System requirements:
  - Any linux system (Ubuntu Desktop latest build preferred)
- Tools:
  - gcc, file, strings, objdump, ltrace,strace installed
  - IDA Free/Binary Ninja/Ghidra Installed (We'll use IDA Free for workshop)
  - Binary Ninja Cloud can also be used (https://cloud.binary.ninja/)
  - gdb + gef (https://github.com/hugsy/gef)
    or pwndbg (https://github.com/pwndbg/pwndbg)

RE Basics Lab Intro/Setup

Lab Instructions & Goals

- CTF styled target
  - Practice target available in git repo (/CTF_Binary/KnightInit)
  - CTF target available as hosted binary later.
- Go through each phase of Reverse Engineering process (each Lab)
- Find and submit flags for each stage

Lab worksheet provided in the git repo

RE Basics Lab - Basic Analysis

Stage 0

Analyse the target file for following:

- Binary type
- Architecture
- Linking
- Stripped/non-stripped
- Hard-coded secrets

Tools

- file
- strings

RE Basics Lab - Static Analysis

Stage 1 & 2

- Disassemble the target using objdump
- Familiarize with objdump output
- Disassemble the target using IDA Free (or Ghidra/Binary Ninja)
- Understand the control flow.

Tools:

- objdump
- IDA Free/Ghidra/Binary Ninja

RE Basics Lab - Dynamic Analysis

Stage 3

- Tracing the target libraries and system calls
- Understanding the control flow of target during runtime
- Quick primer on and familiarizing with gdb

Tools:

- ltrace
- strace
- gdb (gef/pwndbg)

CTF

CTF

04.

Reverse Engineering - CTF

STC_50f5a4c7355fb82a67b3c18624e837e2

RE CTF

Stage 4 onward..

- The CTF Binary is hosted on:

<IP>:1337

- Only the first hardcoded **flag** is same for local and hosted binary.
- Get the real flags from hosted binary.

Total Flags : 5

- Use the gained knowledge to find more flags in the target
- Submit the flags via Twitter or Discord DM.

More Learning RESOURCES & References

- PWN College - https://pwn.college/
- @mytechnotalent: https://0xinfection.github.io/reversing/

  https://github.com/mytechnotalent/Reverse-Engineering

- Practice RE with crackmes: https://crackmes.one/

THANK YOU!

Reach out to me at:

Twitter: @samanl33t

Email: saman.J.L33T@gmail.com

THANKS!

Do you have any questions?
youremail@freepik.com
+91 620 421 838
yourcompany.com

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik

Please keep this slide for attribution

Marketing plan

Marketing plan