



ELF Binary Obfuscation



Index

- **ELF Structure**

- Overview
- ELF Header
- Sections & Segments
- ASLR
- Dynamic Relocations

- **0Pack**

- The beginnings
- 0Pack in action
- So what happened?
- First idea
- Solution
- 0Pack inner workings

Overview

```
~$uname -m
i686
~$./simple.el
Hello World!
```

Header^{1/2}

technical details for
identification and execution

simple.elf

sections

contents of the executable

header^{2/2}

ELF header

Program Header table

Code

Data

Sections' names

Section Header table

ELF Header

```
typedef struct
{
    unsigned char    e_ident[EI_NIDENT];    /* Magic number and other info */
    Elf64_Half       e_type;                /* Object file type */
    Elf64_Half       e_machine;             /* Architecture */
    Elf64_Word       e_version;             /* Object file version */
    Elf64_Addr       e_entry;               /* Entry point virtual address */
    Elf64_Off        e_phoff;               /* Program header table file offset */
    Elf64_Off        e_shoff;               /* Section header table file offset */
    Elf64_Word       e_flags;               /* Processor-specific flags */
    Elf64_Half       e_ehsize;              /* ELF header size in bytes */
    Elf64_Half       e_phentsize;           /* Program header table entry size */
    Elf64_Half       e_phnum;              /* Program header table entry count */
    Elf64_Half       e_shentsize;           /* Section header table entry size */
    Elf64_Half       e_shnum;              /* Section header table entry count */
    Elf64_Half       e_shstrndx;            /* Section header string table index */
} Elf64_Ehdr;
```

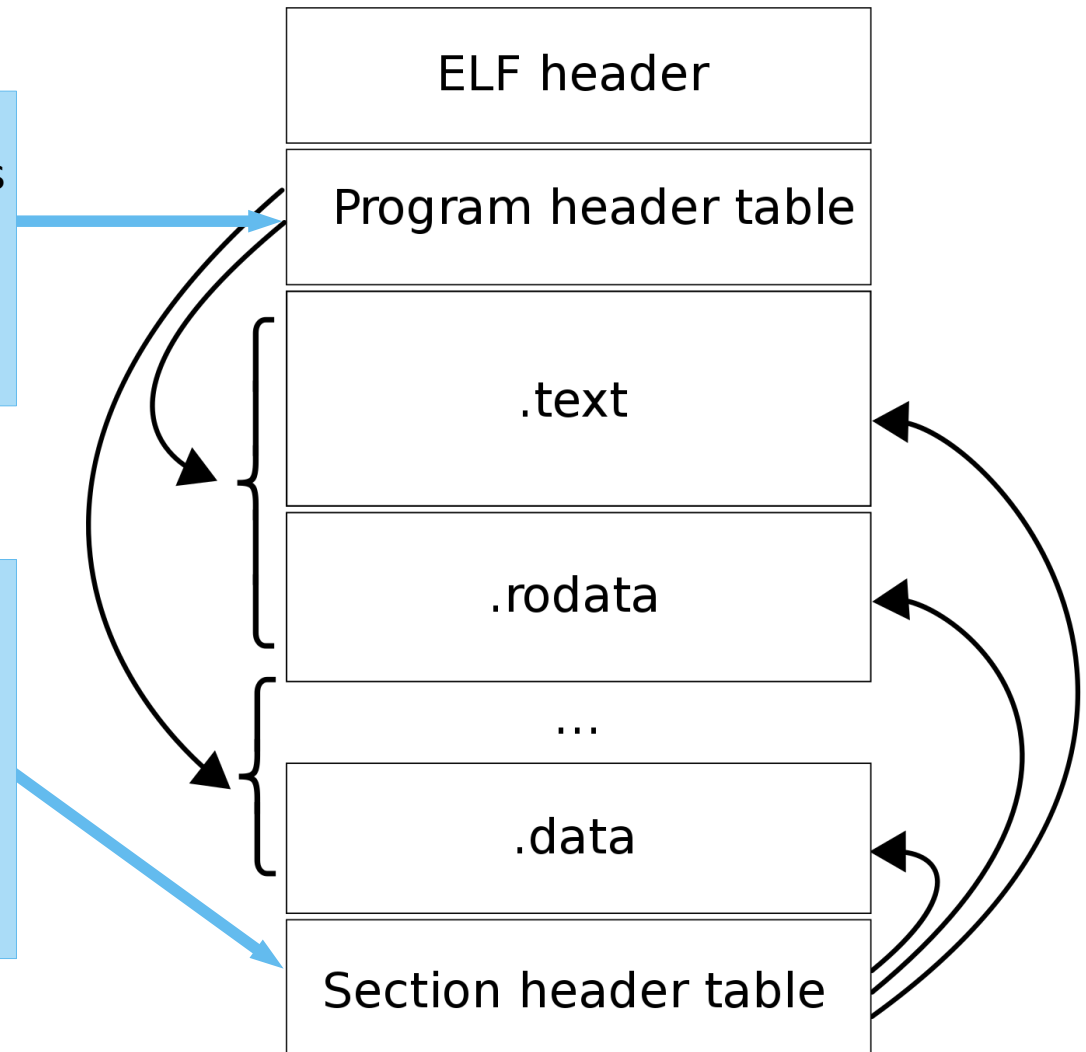
Sections & Segments

Segments

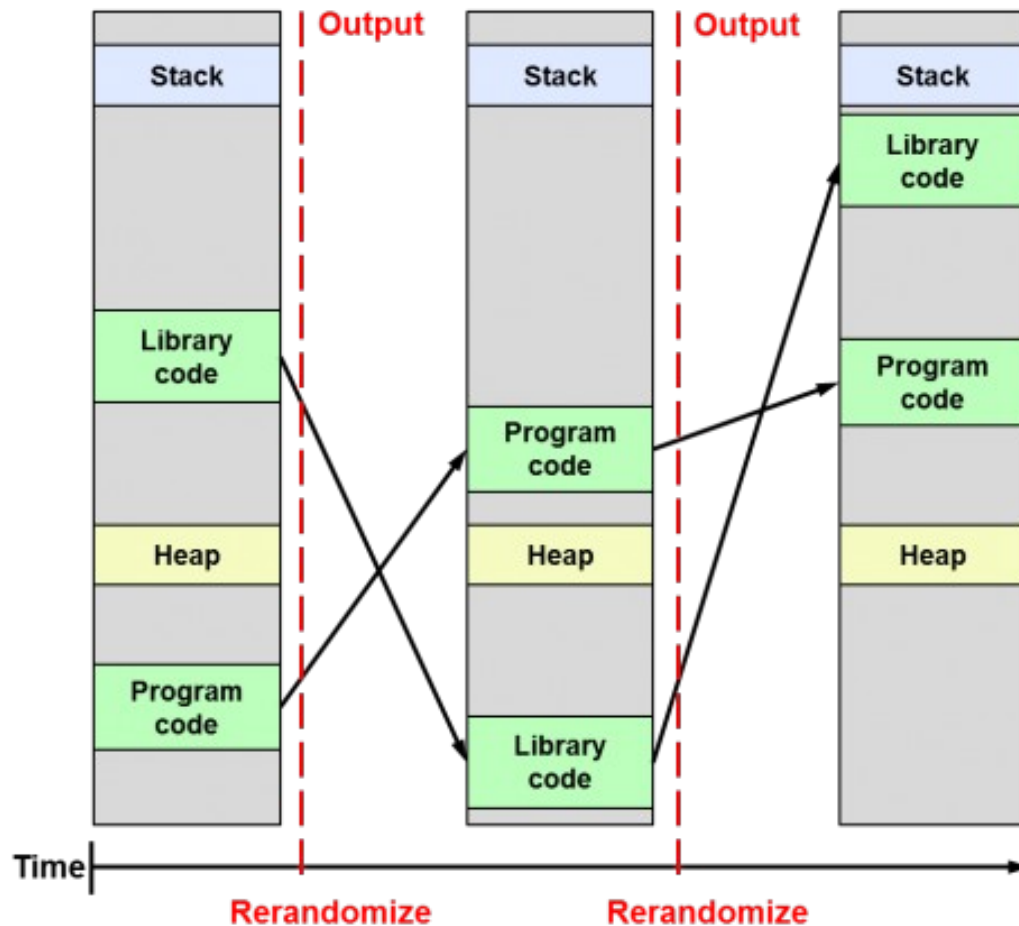
- Equals to one or more memory pages
- Permissions located in program header table

Sections

- Purely optional
- Metadata to divide a segment
- Can be loaded into memory but doesn't have to



Address Space Layout Randomization



Randomized base addresses of shared libraries and segments of the binary and stack

Address Space Layout Randomization



Dynamic Relocations

Normal x64 relocation struct

```
4 typedef struct {  
5     uint64_t r_offset;  
6     uint64_t r_info;  
7 } Elf64_Rel;
```

Relocation struct with addend

```
4 typedef struct {  
5     uint64_t r_offset;  
6     uint64_t r_info;  
7     int64_t r_addend;  
8 } Elf64_Rela;
```

- Relocations patch the binary on load time
- Many different relocation types exists
- Only a few ever get used
- The other ones are mostly unknown

Dynamic Relocations

Relocation Table Entry Format

| Description | offset | size | field name |
|--------------------------------|--------|------|------------|
| Offset used to calculate reloc | 0x00 | 8 | r_offset |
| Reloc typing meta-data | 0x08 | 8 | r_info |
| Extra argument used in reloc | 0x10 | 8 | r_addend |

| Description | name |
|-----------------------------------|----------------|
| Section for holding plt relocs | .rela.plt |
| Section for dynamic symbol relocs | .rela.dyn |
| | .rel(a).[name] |

| Macro | |
|----------------------|-------------------------|
| ELF64_R_SYM(r_info) | ((r_info) >> 32) |
| ELF64_R_TYPE(r_info) | ((r_info) & 0xffffffff) |

| index | Symbol |
|-------|------------|
| 0 | function_1 |
| 1 | function_2 |
| ... | ... |

| Code | field name | Description |
|------|--------------------|---------------------------------|
| 0x00 | R_X86_64_NONE | No Reloc |
| 0x01 | R_X86_64_64 | Processor Specific Hidden class |
| .. | ... | ... |
| 0x06 | R_X86_64_GLOB_DAT | Make a GOT entry |
| 0x07 | R_X86_64_JUMP_SLOT | Make a PLT entry |
| 0x08 | R_X86_64_RELATIVE | Adjust reloc by program base |
| .. | ... | ... |

Note: Upper 4 bytes of r_info are the index, the lower 4 bytes are for the type

The beginnings

My goal:

Building a simple ELF obfuscator for ASLR binaries...

Which led to:

Oh nooo,
My debugger broke (◡_◡)

0Pack in action



So what happened here?

Entrypoint is **0**

Debugger checks if **entrypoint** is **NULL**

If true

They **parse** the **symbol table** to locate the main function

Sets the entrypoint to the **main** function

The debugger sets the **breakpoint** at the **wrong address!**

Symbol Table

| | Name | Binding | Type | Address |
|---|-----------|---------|----------|---------|
| 1 | "foo_bar" | Global | Data | 0x80 |
| 2 | "do_it" | Global | Function | 0x1000 |
| 3 | "func_X" | Local | Function | 0x2000 |
| 4 | "func_a" | Global | Function | UNDEF |

global variable

global function

static function

external function

Relocation Table

| Info | Offset |
|------|--------|
| 1 | 0x100 |
| 1 | 0x180 |
| 2 | 0x2044 |
| 3 | 0x1200 |
| 3 | 0x2200 |
| 3 | 0x4300 |
| 4 | 0x5000 |

Executable Binary Code

foo_bar = 10;

func_X (1, 2);

So how do we get a working executable out of this?

Execution now
starts here



```
typedef struct
{
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half       e_type;
    Elf64_Half       e_machine;
    Elf64_Word       e_version;
    Elf64_Addr       e_entry;
    Elf64_Off        e_phoff;
    Elf64_Off        e_shoff;
    Elf64_Word       e_flags;
    Elf64_Half       e_ehsize;
    Elf64_Half       e_phentsize;
    Elf64_Half       e_phnum;
    Elf64_Half       e_shentsize;
    Elf64_Half       e_shnum;
    Elf64_Half       e_shstrndx;
} Elf64_Ehdr;
```

First idea

ELF Header seen as code

```
=< 0x00000000 7f45      jg 0x47
0x00000002 4c460201    add r8b, byte [rcx]
0x00000006 0100      add dword [rax], eax
0x00000008 0000      add byte [rax], al
0x0000000a 0000      add byte [rax], al
0x0000000c 0000      add byte [rax], al
0x0000000e 0000      add byte [rax], al
0x00000010 0300      add eax, dword [rax]
0x00000012 3e0001    add byte ds:[rcx], al
0x00000015 0000      add byte [rax], al
0x00000017 00e0      add al, ah
0x00000019 06      invalid
0x0000001a 0000      add byte [rax], al
0x0000001c 0000      add byte [rax], al
0x0000001e 0000      add byte [rax], al
0x00000020 400000    add byte [rax], al
0x00000023 0000      add byte [rax], al
0x00000025 0000      add byte [rax], al
0x00000027 00602f    add byte [rax + 0x2f], ah
0x0000002a 0000      add byte [rax], al
0x0000002c 0000      add byte [rax], al
0x0000002e 0000      add byte [rax], al
0x00000030 0000      add byte [rax], al
0x00000032 0000      add byte [rax], al
0x00000034 400038    add byte [rax], dil
0x00000037 000a      add byte [rdx], cl
0x00000039 004000    add byte [rax], al
0x0000003c 2300      and eax, dword [rax]
0x0000003e 2200      and al, byte [rax]
;-- segment.PHDR:
0x00000040 06      invalid
0x00000041 0000      add byte [rax], al
0x00000043 000400    add byte [rax + rax], al
0x00000046 0000      add byte [rax], al
0x00000048 400000    add byte [rax], al
```

0x7f45 => jg 0x47

- ELF magic translates to indirect jump
- Jumps into program header
- Change program header value to another jump?



Solution

Table 4.10: Relocation Types

| Name | Value | Field | Calculation |
|--------------------|-------|--------|-------------|
| R_X86_64_NONE | 0 | none | none |
| R_X86_64_64 | 1 | word64 | $S + A$ |
| R_X86_64_PC32 | 2 | word32 | $S + A - P$ |
| R_X86_64_GOT32 | 3 | word32 | $G + A$ |
| R_X86_64_PLT32 | 4 | word32 | $L + A - P$ |
| R_X86_64_COPY | 5 | none | none |
| R_X86_64_GLOB_DAT | 6 | word64 | S |
| R_X86_64_JUMP_SLOT | 7 | word64 | S |

Writes the
Symbol value + Addend
to the target location

The symbol value can be set
to 0

Relocations now
can overwrite any bytes!

Relocation struct with addend

```
4 typedef struct {
5     uint64_t r_offset;
6     uint64_t r_info;
7     int64_t r_addend;
8 } Elf64_Rela;
```

0Pack inner workings

Set entrypoint to 0

Set first segment to RWX

Append code that erases shellcode after execution

Append a relative jmp to original entrypoint

Convert shellcode to R_X86_64_64 Relocations



Any Questions?