Constructing ELF Metadata
BerlinSides 0x3
27 May 2012

Rebecca Shapiro and Sergey Bratus
Dartmouth College
This Talk in One Minute

- “Deep magic” before a program can run
  - ELF segments, loading, relocation,
- “Deeper magic” to support dynamic linking
  - Dynamic symbols, loading of libraries
- Many pieces of code – enough to **program anything** (Turing-complete)
  - In perfectly **valid** ELF **metadata** entries alone
- Runs before most **memory protections** are set for the rest of runtime
- Runs with access to **symbols** (ASLR? what ASLR?)
The Quest

- ELF background
- Prior work with abusing ELF
- Everything you need to know about ELF metadata for this talk
- Branfuck to ELF compiler
- Relocation entry backdoor
  - Demo exploit
ELF
Executable and Linking Format

- How gcc toolchain components communicate
  - Assembler
  - Static link editor
  - Runtime link editor (RTLD)
  - Dynamic loader
ELF Components

- 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 Section

- All data/code is contained in ELF sections
  - Except ELF, section, and segment headers

- 1 section --> 1 section header
  - Describes type, size, file offset, memory offset, etc, for linker/loader

- Most sections contain one of
  - Table of a single type of metadata
  - Null terminated strings
  - Mixed data (ints, long, etc)
  - Code
Sections of interest

- Symbol table (.dynsym)
- Relocation tables (.rela.dyn, .rela.plt)
- Global offset talbe (.got)
- Procedure linkage table (.got.plt)
- Dynamic table (.dynamic)
Symbol table

- 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>00000000000376d98</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL DEFAULT</td>
<td>31</td>
<td>stdin</td>
<td></td>
</tr>
<tr>
<td>7408</td>
<td>00000000000525c0</td>
<td>42</td>
<td>FUNC</td>
<td>GLOBAL DEFAULT</td>
<td>12</td>
<td>putc</td>
<td></td>
</tr>
</tbody>
</table>

- Symbol definition for 64-bit architecture:

```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 Entry

- **Where to write what value at load/link time**
- **For amd64:**
  ```c
  typedef struct {
      Elf64_Addr r_offset;
      uint64_t   r_info;
      int64_t    r_addend;
  } Elf64_Rela;
  ```

- **r_info:**
  - Relocation entry type
    ```c
    #define ELF64_R_TYPE(i) ((i) & 0xffffffff)
    ```
  - Associated symbol table entry index
    ```c
    #define ELF64_R_SYM(i) ((i) >> 32)
    ```
  - amd64 ABI defines 37 relocation types
  - gcc toolchain uses 13 types (1 not in ABI)
GOT and PLT
Global Offset Table and Procedure Linkage Table

- Each function requiring dynamic linking has an entry in each
- GOT is a table of addresses
- GOT[1] = object's link_map struct
  - Data on ELF objects used by RTLD/linker
- GOT[2] = &_dl_fixup (dynamic linker function)
- GOT entry for function is pointer to function or code in PLT that calls _dl_fixup
- PLT is code that works with GOT to run dynamic linker if needed
Dynamic section

- Table of metadata used by runtime loader

```c
typedef struct {
    Elf64_Sxword   d_tag;
    union {
        Elf64_Xword d_val;
        Elf64.Addr  d_ptr;
    } d_un;
} Elf64_Dyn;
```

- Types of interest
  - DT_RELJA, DT_RELASZ
  - DT_RELACOUNT
  - DT_SYM
  - DT_JMPREL, DT_PLTRELSZ
Interesting dynamic section entries

- **DT_RELACOUNT**
  - Start of .rela.dyn table, size, and number of entries of type R_*_RELATIVE

- **DT_SYM**
  - Location of symbol table (.dynsym)

- **DT_JMPREL, DT_PLTRELSZ**
  - Location of .rela.plt table
    - relocation entries processed by dynamic loader
  - Size of .rela.plt table
The story of exec

exec(ping)

ping
The story of exec
The story of exec
Memory layout of ping (abbrev)

- 00400000-00408000 r-xp ping
- 00607000-00608000 r--p ping
- 00608000-00609000 rw-p ping
- 00609000-0061c000 rw-p
- 02165000-02186000 rw-p [heap]
- 7fc2224d2000-7fc2224de000 r-xp libnss_files-2.13.so
- 7fc2226dd000-7fc2226de000 r--p libnss_files-2.13.so
- 7fc2226de000-7fc2226df000 rw-p libnss_files-2.13.so
- 7fc2226df000-7fc222876000 r-xp libc-2.13.so
- 7fc222a75000-7fc222a79000 r--p libc-2.13.so
- 7fc222a79000-7fc222a7a000 rw-p libc-2.13.so
- 7fc222a7a000-7fc222a80000 rw-p
- 7fc222a80000-7fc222aa1000 r-xp ld-2.13.so
- 7fc222c77000-7fc222c7a000 rw-p
- 7fc222c9d000-7fc222ca0000 rw-p
- 7fc222ca0000-7fc222ca1000 r--p ld-2.13.so
- 7fc222ca1000-7fc222ca3000 rw-p ld-2.13.so
- 7fff01379000-7fff0139a000 rw-p [stack]
General process memory layout

- executable
- heap
- dynamic library 0
- dynamic library n
- libc.so
- ld.so (linker/loader)
- stack
General process memory layout
A processes' segments

executable

0x40000
• string table (.dynstr)
• symbol table (.dynsym)
• relocation tables (.rela.dyn, .rela.plt)
• code (.plt, .init, .text, .fini)

0x40800
0x60700
read/execute

0x60800
read only

0x60900
read/write

0x60800
• dynamic table (.dyn)
• global offset table (.got, .got.plt)
• data (.data, .bss)

00400000-00408000 r-xp 00000000 08:06 261244 /bin/ping
00607000-00608000 r--p 00007000 08:06 261244 /bin/ping
00608000-00609000 rw-p 00008000 08:06 261244 /bin/ping
General process memory layout

The executable: our perspective*

- Symbol table
- Relocation entries
- PLT
- Code
- Got
- Data

Libc... interesting code dwells here

Linker/loader

ld.so's data and heap metadata to process loaded ELF objects
General process memory layout

the executable: our perspective

symbol table
relocation entries
plt
code

got
data

ld.so's data and heap
metadata to process loaded ELF objects

executable

libc... interesting code dwells here

linker/loader
struct link_map {
    ElfW(Addr) l_addr;     /* Base address shared object is loaded at. */
    char *l_name;          /* Absolute file name object was found in. */
    ElfW(Dyn) *l_ld;       /* Dynamic section of the shared object. */
    struct link_map *l_next, *l_prev; /* Chain of loaded objects. */
}

struct libname_list *l_libname
...
ElfW(Dyn) *l_info[DT_NUM + DT_THISPROCNUM + DT_VERSIONTAGNUM]
...
union {
    const Elf32_Word *l_gnu_chain_zero;
    const Elf_Symndx *l_buckets;
};

unsigned int l_direct_opencount; /* Reference count for dlopen/dlclose. */
enum {                           /* Where this object came from. */
    lt_executable,            /* The main executable program. */
    lt_library,               /* Library needed by main executable. */
    lt_loaded                 /* Extra run-time loaded shared object. */
} l_type:2;

unsigned int l_relocated:1;    /* Nonzero if object's relocations done. */
...

size_t l_relo_size;
...
Fun ways to abuse ELF metadata

- Change entrypoint to point to injected code
- Inject object files (*mayhem, phrack 61:8*)
- Intercept library calls to run injected code
  - Injected in executable
    - Cesare PLT redirection (*Phrack 56:7*)
    - Mayhem ALTPLT (*Phrack 61:8*)
  - Resident in attacker-built library
    - LD_PRELOAD (example: *Jynx-Kit* rootkit)
    - DT_NEEDED (Phrack 61:8)
    - Loaded at runtime (*Cheating the ELF, the grugq*)
  - Injected in library
- LOCERRYATE (*Skape, Uniformed 2007*)
  - Unpack binaries using relocation entries
More fun with relocation entries

Warning. The following you are about to see is architecture and libc implementation dependant. Please try this at home, but there are no guarantees it will work with your architecture/gcc toolchain combination. (Ours is Ubuntu 11.10's eglibc-2.13 on amd64)

Not all Brainfuck instructions work with ASLR.
Injecting Relocation/Symbol tables

- Use eresi toolkit
- Injects into executable's r/w segment
typedef struct {
    Elf64_Addr r_offset;
    uint64_t r_info; // contains type and symbol number
    int64_t r_addend;
} Elf64_Rela;

- Let \( r \) be our `Elf64_Rela`, \( s \) be the corresponding `Elf64_Sym` (if applicable)
- **R_X86_64_COPY**
  - \( \text{memcpy}(r.r\_offset, s.st\_value, s.st\_size) \)
- **R_X86_64_64**
  - \( *(\text{base}+r.r\_offset) = s.st\_value +r.r\_addend+\text{base} \)
- **R_X86_64_32**
  - Same as _64, but only writes 4 bytes
- **R_X86_64_RELATIVE**
  - \( *(\text{base}+r.r\_offset = r.r\_addend+\text{base}) \)
Symbols of type STT_IFUNC are special!
- `st_value` treated as a function pointer

```c
#include <stdio.h>
int foo (void) __attribute__ ((ifunc ("foo_ifunc")))
static int global = 1;
static int f1 (void) { return 0; }
static int f2 (void) { return 1; }
void *foo_ifunc (void) { return global == 1 ? f1 : f2; }
int main () { printf (%d\n", foo()); }
```

Symbols:
```
000000000040053a <foo_ifunc>:
```
```assembly
   ....
40053e:  8b 05 e4 0a 20 00   mov  0x200ae4(%rip),%eax    # 601028 <global>
400544:  83 f8 01           cmp   $0x1,%eax
400547:  75 07             jne  400550 <foo_ifunc+0x16>
   ....
400550:  b8 2f 05 40 00     mov   $0x40052f,%eax
400555:  5d                pop    %rbp
400556:  c3                retq
```
Brainfuck Primer

6 instructions:
1) > Increment the pointer.
2) < Decrement the pointer.
3) + Increment the byte at the pointer.
4) - Decrement the byte at the pointer.
5) [ Jump forward past the matching ] if the byte at the pointer is zero.
6) ] Jump backward to the matching [ unless the byte at the pointer is zero.
7) . Output the byte at the pointer.
8) , Input a byte and store in byte at the pointer.

Source: http://www.muppetlabs.com/~breadbox/bf/
Brainfuck Primer

- 6 instructions:
  1) > Increment the pointer.
  2) < Decrement the pointer.
  3) + Increment the byte at the pointer.
  4) - Decrement the byte at the pointer.
  5) [ Jump forward past the matching ] if the byte at the pointer is zero.
  6) ] Jump backward to the matching [ unless the byte at the pointer is zero.
  7) . Output the byte at the pointer.
  8) , Input a byte and store in byte at the pointer.

Source: http://www.muppetlabs.com/~breadbox/bf/
Brainfuck Primer
Hello, World

// Hello World in brainfuck
// Creds to Speedy
>+++++++++++[<++++++++++++>-]<.>+++++++++
[<++++>.-]<+.++++++++++.+++.[-]
>++++++++++[<++++>-] <.++++++++++++++
[<++++++++++>-] <-.--------.+++ 
.--------.--------.[-] >++++++[<++++>- ]<+.[-]
++++++++++++++.

Source: www.helloworld.org
**ELF Brainfuck Setup**

<table>
<thead>
<tr>
<th>.dynsym table</th>
<th>.rela.dyn table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty)</td>
<td>Brainfuck instruction 0</td>
</tr>
<tr>
<td>Original dynsym 0</td>
<td>...</td>
</tr>
<tr>
<td>Original dynsym 1</td>
<td>Brainfuck instruction n</td>
</tr>
<tr>
<td>...</td>
<td>&quot;Code&quot; that cleans up some link_map data</td>
</tr>
<tr>
<td>Original dynsym n</td>
<td>&quot;Code&quot; that forces branch to next reloc entry</td>
</tr>
<tr>
<td>Address tape head is pointing at</td>
<td>&quot;Code&quot; that finishes cleaning link_map</td>
</tr>
<tr>
<td>Copy of tape head's value</td>
<td>Original .rela.dyn entry 0</td>
</tr>
<tr>
<td>IFUNC that always returns 0</td>
<td>...</td>
</tr>
<tr>
<td>Copy of IFUNC address</td>
<td>Original .rela.dyn entry m</td>
</tr>
</tbody>
</table>
ELF Brainfuck Setup

- Data needed at compile time
  - Address of executable's link_map
    - In future versions, get this automatically
  - Address of instructions that return 0
    - ROP-style
  - Stack location
  - Location in memory of executable's:
    - DT_RELAX
    - DT_RELASZ
    - DT_SYM
    - DT_JMPREL
    - DT_PLTREL
    - Collected at runtime (compile time?)

- Compiler works with existing executable
ELF Brainfuck Tape Pointer

- Relocation/symbol entries must be in writable memory
- Tape must be in writable memory
ELF Brainfuck Tape Pointer

<table>
<thead>
<tr>
<th>Address tape head is pointing at</th>
<th>t_ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy of tape head's value</td>
<td>t_val</td>
</tr>
</tbody>
</table>

```
0xb33f0000
0xb33f0008
0xb33f0010
0xb33f0018
0xb33f0020
```

```
0x0000000000000000
0x0000000000000001
0x0000000000000000
0x0000000000000000
0x0000000000000000
```
ELF Brainfuck Tape Pointer

\[mv\_ptr = \{\text{offset=}&(t\_ptr.value), \text{type} = 64, \text{sym=t\_ptr, addend}=8*n\}\]

\[copy\_val = \{\text{offset=}&(t\_val.value), \text{type} = \text{COPY}, \text{sym=p\_tptr}\}\]
mv_ptr = {offset=&(t_ptr.value), type = 64, sym=t_ptr, addend=8}
copy_val = {offset=&(t_val.value), type = COPY, sym=p_ptr}

n=1

<table>
<thead>
<tr>
<th>t_ptr</th>
<th>0xb33f0008</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_val</td>
<td>0</td>
</tr>
</tbody>
</table>

0xb33f0000  0x0000000000000000
0xb33f0008  0x0000000000000001
0xb33f0010  0x0000000000000000
0xb33f0018  0x0000000000000000
0xb33f0020  0x0000000000000000
mv_ptr = \{offset=&(t_ptr.value), type = 64, sym=t_ptr, addend=8\}
copy_val = \{offset=&(t_val.value), type = COPY, sym=p_tptr\}
add = {offset=&(t_ptr.value), type = 64, sym=t_val, addend=n}
get_ptr = {offset=&(update.offset), type = 64, sym=t_ptr}
update = {offset=?????, type = 64, sym=t_val}
add = \{offset=&(t\_ptr.value), type = 64, sym=t\_val, addend=2\}
get\_ptr = \{offset=&(update.offset), type = 64, sym=t\_ptr\}
update = \{offset=?????, type = 64, sym=t\_val\}
add = {offset=&(t_ptr.value), type = 64, sym=t_val, addend=2}
get_ptr = {offset=&(update.offset), type = 64, sym=t_ptr}
update = {offset=0xb33f0008, type = 64, sym=t_val}
add = {offset=&(t_ptr.value), type = 64, sym=t_val, addend=2}
get_ptr = {offset=&(update.offset), type = 64, sym=t_ptr}
update = {offset=0xb33f0008, type = 64, sym=t_val}
ELF BF Unconditional Branch/Loop

- How relocation entries get 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 != &GL(dl_rtld_map))
            _dl_relocate_object (l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0, consider_profiling);

        ... 
        l = l->l_prev;
    }
```

TODO:
- set l->l_prev = l
ELF Brainfuck Unconditional Branch

- How relocation entries get 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 != &GL(dl_rtld_map))
        _dl_relocate_object (l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0, consider_profiling);

    ... 
    l = l->l_prev;
}
while (l);
```

TODO:
- set l->l_prev = l
void _dl_relocate_object (struct link_map *l, struct r_scope_elem *scope[],
   int reloc_mode, int consider_profiling)
{

    if (l->l_relocated)
        return;

    ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);

    /* Mark the object so we know this work has been done. */
    l->l_relocated = 1;

    /* In case we can protect the data now that the relocations are
done, do it. */
    if (l->l_relro_size != 0)
        _dl_protect_relro (l);

    ...

    TODO:
    - set l->l_prev = l
    - fix l->l_relocated
ELF Brainfuck Unconditional Branch

- How relocation entries get processed

```c
void _dl_relocate_object (struct link_map *l, struct r_scope_elem *scope[],
    int reloc_mode, int consider_profiling)
{
    if (l->l_relocated)
        return;
    ...
    ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
    ...
    /* Mark the object so we know this work has been done. */
    l->l_relocated = 1;
    ...
    /* In case we can protect the data now that the relocations are done, do it. */
    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
ELF Brainfuck Unconditional Branch

- How relocation entries get processed

```c
do
{
    struct libname_list *Inp = l->l_libname->next;

    while (__builtin_expect (Inp != NULL, 0))
    {
        Inp->dont_free = 1;
        Inp = Inp->next;
    }

    if (l != &GL(dl_rtld_map))
        _dl_relocate_object (l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
            consider_profiling);

    ...  
    l = l->l_prev;
} while (l);
```
ELF Brainfuck Unconditional Branch Bookeeping

- Fix l->l_relocated
- Set l->l_prev = l
- Set l->l_relo_size = 0
- Set l->l_info[DT_RELA] = &next rel to process
- Fix l->l_info[DT_RELASZ]
ELF Brainfuck Unconditional Branch Bookkeeping

- Fix l->l_relocated
  - {offset =&(l->l_buckets), type = RELATIVE, addend=0}
  - {offset =&(l->l_direct_opencount), type = RELATIVE, addend=0}
  - {offset =&(l->l_libname->next), type = RELATIVE, addend=&(l->l_relocated) + 4*sizeof(int)}

- Set l->l_prev = l
  - {offset =&(l->l_prev), type = RELATIVE, addend=&l}

- Set l->l_relo_size = 0
  - (etc)

- Set l->l_info[DT_RELA] = &next rel to process
- Fix l->l_info[DT_RELASZ]
end is on stack, set it to 0

{offset = &end, type = RELATIVE, addend=0}
ELF Brainfuck Conditional Branches

- Perform all branch book keeping
- IFUNC symbol only processed as function if `st_shndx != 0`

.dynsym table

- (empty)
- Original dynsym 0
- Original dynsym 1
- ...
- Original dynsym n
- Address tape head is pointing at
- Copy of tape head's value
- IFUNC that always returns 0
- Copy of IFUNC address
ELF Brainfuck Conditional Branches

setifunc = {offset=&(ifunc.shndx), type = 32, sym=t_val}*
get_ptr = {offset=&(ifunc.value), type = 32, sym=ifunc_c}
update = {offset=&end, type = 64, sym=ifunc}

*BUG ALERT: A symbol's st_shndx is only 2 bytes long. Tape entries need to be 2 bytes long or less for this to behave as expected. This will be fixed in future versions of the bf compiler.
ELF Brainfuck Conditional Branches

setifunc = {offset=&(ifunc.shndx), type = 32, sym=t_val}
get_ptr = {offset=&(ifunc.value), type = 32, sym=ifunc_c}
update = {offset=&end, type = 64, sym=ifunc}
ELF Brainfuck Conditional Branches

```
setifunc = {offset=&(ifunc.shndx), type = 32, sym=t_val}
get_ptr = {offset=&(ifunc.value), type = 32, sym=ifunc_c}
update = {offset=&end, type = 64, sym=ifunc}
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_val</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ifunc</td>
<td>0x1</td>
<td>&amp;return0</td>
</tr>
<tr>
<td>ifunc_c</td>
<td>0</td>
<td>&amp;return0</td>
</tr>
</tbody>
</table>

```
0xb33f0000 0x000000000000000000000001
```

end=0
ELF Brainfuck Conditional Branches

setifunc = {offset=&(ifunc.shndx), type = 32, sym=t_val}
get_ptr = {offset=&(ifunc.value), type = 32, sym=ifunc_c}
update = {offset=&end, type = 64, sym=ifunc}

<table>
<thead>
<tr>
<th></th>
<th>t_val</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifunc</td>
<td>0</td>
<td>&amp;return0</td>
<td></td>
</tr>
<tr>
<td>ifunc_c</td>
<td>0</td>
<td>&amp;return0</td>
<td></td>
</tr>
</tbody>
</table>

0xb33f0000

end=&return0

(>> relocation entries)
ELF Brainfuck ']

- "Jump forward past the matching ] if the byte at the pointer is zero"
- Prepare for branch, set branch location to & of entry after unconditional branch
- If value == 0, run unconditional branch:
  - Branch past ']
- If value != zero
  - We have skipped over unconditional branch
  - Continue to process relocation entries after ']'
ELF Brainfuck ']'

- "Jump backward to the matching [ unless the byte at the pointer is 0"
- Prepare for branch, set branch location to & of relocation entry after ']'  
- If value == 0, continues processing  
- If value != zero
  - Stops processing relocation entries, branch executes
Implementation Notes

- Used eresi toolchain to inject/edit metadata
- Injects metadata into r/w section
- More bookkeeping is necessary to ensure executable works (not mentioned in talk)
- Code coming soon to a github near you
  - elf-bf-tools repository on github
More fun with relocation entries:
I'm My Own Grandpa

[Diagram showing a family tree with labels such as Ray's Father, Widow, Baby Boy, Red Head, and Uncle.]
Following a pointer

- To get linkmap->l_next->l_addr:
- Store &got+0x8 in a symbol (DT_PLTGOT value)

Symbols:

symgot = {value: &got+8, size: 8, ...}

- Use the following relocation entries with that symbol

Relocation entries:

get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}
Following a pointer

\[
\text{symgot} = \{\text{value} &: \text{got\_0x8}, \text{size} : 8, \ldots\}
\]

\[
\text{get\_exec\_linkmap} = \{\text{offset} &: (\text{symgot\_value}), \text{type} = \text{COPY}, \text{sym} = 0\}
\]

\[
\text{get\_l\_next} = \{\text{offset} &: (\text{symgot\_value}), \text{type} = 64, \text{sym} = 0, \text{addend} = 0x18\}
\]

\[
\text{deref\_l\_next} = \{\text{offset} &: (\text{symgot\_value}), \text{type} = \text{COPY}, \text{sym} = 0\}
\]

\[
\text{get\_l\_addr} = \{\text{offset} &: (\text{symgot\_value}), \text{type} = \text{COPY}, \text{sym} = 0\}
\]
Following a pointer

`symgot = {value:&got_0x8, size: 8, ...}`

`get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}`

`get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}`

`deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}`

`get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}`
Following a pointer

```c
symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}

write
```

![Diagram](image_url)
Following a pointer

symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next={offset=&(symgot.value),type = 64,sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}
Following a pointer

symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next={offset=&(symgot.value),type = 64,sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}

write

get_l_next

&linkmap->l_next
Following a pointer

symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}

deref_l_next

&l_next

calculate
Following a pointer

symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}
write

deref_l_next

l_next
Following a pointer

```plaintext
symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}
```

write

deref_l_next  l_next
Following a pointer

symgot = {value:&got_0x8, size: 8, ...}
get_exec_linkmap = {offset=&(symgot.value), type = COPY, sym=0}
get_l_next = {offset=&(symgot.value), type = 64, sym=0, addend=0x18}
deref_l_next = {offset=&(symgot.value), type = COPY, sym=0}
get_l_addr = {offset=&(symgot.value), type = COPY, sym=0}
Following a pointer

symgot = \{ value:&got_0x8, size: 8, ... \}
get_exec_linkmap = \{ offset=&(symgot.value), type = COPY, sym=0 \}
get_l_next = \{ offset=&(symgot.value), type = 64, sym=0, addend=0x18 \}
deref_l_next = \{ offset=&(symgot.value), type = COPY, sym=0 \}
get_l_addr = \{ offset=&(symgot.value), type = COPY, sym=0 \}

write

symgot's value is base address of some ELF object
Demo exploit

- Built backdoor into Ubuntu's inetutils v1.8 ping
- Ping runs suid as root
- Given ”-t <string>”
  - -t, --type=TYPE send TYPE packets
  - if (strcasecmp (<string>, "echo") == 0) ....

- Goals:
  - Redirect call to strcasecmp to execl
  - Prevent call to setuid that drops root privs
  - Work in presence of library ASLR
Demo exploit

- **Goals:**
  - Redirect call to strcasecmp to execl
    - Set strcasecmp's GOT entry to &execl
  - Prevent privilege drop
    - Set setuid's GOT entry to & retq instruction
  - Found offset to exel and a retq instruction in glibc
  - Need to find base address of glibc @ runtime
  - Use link_map traversal trick!
    - The rest is simple addition/relocation
(video of demo was here)
Thanks!

- Sergey Bratus
- Sean Smith

**Inspirations:**
- The gruggq
- ERESI and Elfsh folks
- mayhem
- Skape
Questions?