Document

Chaining return-into-libc calls

1. “esp lifting” method
2. Frame faking
3. Inserting null bytes

PaX features

1. PaX basics
2. PaX and return-into-libc exploits
3. PaX and mmap base randomization

The dynamic linker’s dl-resolve() function

1. A few ELF data types
2. A few ELF data structures
3. How dl-resolve() is called from PLT

Defeating PaX

1. Requirements
2. Building the exploit

Misc

1. Portability
2. Other types of vulnerabilities
3. Other non-exec solutions
4. Improving existing non-exec schemes

5. The versions used

Practice

Babystack from 0ctf 2018

PREPARE:

- JMPREL
- SYMTAB
- STRTAB

```
STRTAB      0x80484f8 the location of string table (type char *)
SYMTAB      0x8048268 the location of symbol table (type Elf32_Sym*)
```

```
typedef uint32_t Elf32_Addr ;
typedef uint32_t Elf32_Word ;
typedef struct
{
   Elf32_Addr r_offset ; /* Address */
   Elf32_Word r_info ; /* Relocation type and symbol index */
} Elf32_Rel ;
#define ELF32_R_SYM(val) ((val) >> 8)
#define ELF32_R_TYPE(val) ((val) & 0xff)
```

```
typedef struct
{
   Elf32_Word st_name ; /* Symbol name (string tbl index) */
   Elf32_Addr st_value ; /* Symbol value */
   Elf32_Word st_size ; /* Symbol size */
   unsigned char st_info ; /* Symbol type and binding */
   unsigned char st_other ; /* Symbol visibility under glibc>=2.2 */
   Elf32_Section st_shndx ; /* Section index */
} Elf32_Sym ;
```

```
// call of unresolved read(0, buf, 0x100)
_dl_runtime_resolve(link_map, rel_offset) {
    Elf32_Rel * rel_entry = JMPREL + rel_offset ;
    Elf32_Sym * sym_entry = &SYMTAB [ ELF32_R_SYM ( rel_entry -> r_info )];
    /* Check… */
    char * sym_name = STRTAB + sym_entry -> st_name ;
    _search_for_symbol_(link_map, sym_name);
    // invoke initial read call now that symbol is resolved
    read(0, buf, 0x100);
}
```

```
The following is the simplified dl-resolve() algorithm:

1) calculate some_func's relocation entry
       Elf32_Rel * reloc = JMPREL + reloc_offset;

2) calculate some_func's symtab entry
       Elf32_Sym * sym = &SYMTAB[ ELF32_R_SYM (reloc->r_info) ];

3) sanity check
         assert (ELF32_R_TYPE(reloc->r_info) == R_386_JMP_SLOT);

4) late glibc 2.1.x (2.1.92 for sure) or newer, including 2.2.x, performs
   another check. if sym->st_other & 3 != 0, the symbol is presumed to have
   been resolved before, and the algorithm goes another way (and probably
   ends with SIGSEGV in our case). We must ensure that sym->st_other &
   3 == 0.

5) if symbol versioning is enabled (usually is), determine the version
table
   index
       uint16_t ndx = VERSYM[ ELF32_R_SYM (reloc->r_info) ];

and find version information
       const struct r_found_version *version =&l->l_versions[ndx];

  where l is the link_map parameter. The important part here is that ndx
must
  be a legal value, preferably 0, which means "local symbol".

6) the function name (an asciiz string) is determined:
       name = STRTAB + sym->st_name;

7) The gathered information is sufficient to determine some_func's address.
   The results are cached in two variables of type Elf32_Addr, located at
   reloc->r_offset and sym->st_value.

8) The stack pointer is adjusted, some_func is called.

Note: in case of glibc, this algorithm is performed by the fixup()
function,
called by dl-runtime-resolve().
```

For demonstration purposes only, let us suppose that:

- JMPREL @ 0x0
- SYMTAB @ 0x100
- STRTAB @ 0x200
- controllable area @ 0x300

We need to craft our Elf32_Rel and Elf32_Sym somewhere within the controllable area and provide a rel_offset such that the resolver reads our special forged structures. Let's suppose that the controllable (stack after pivotation ???) are has the following layout.

```
             +--------+
r_offset     |GOT     |  0x300
r_info       |0x2100  |  0x304
alignment    |AAAAAAAA|  0x308
st_name      |0x120   |  0x310
st_value     |0x0     |
st_size      |0x0     |
others       |0x12    |
sym_string   |"syst   |  0x320
             |em\x00" |
             +--------+
```

When _dl_runtime_resolve (link_map , 0x300) is called, the 0x300 offset is used to get the Elf32_Rel* rel = JMPREL + 0x300 == 0x300.

Secondly, the Elf32_Sym is accessed using the r_info field from 0x304.

Elf32_Sym* sym = &SYMTAB[(0x2100 >> 8)] == 0x310.

The last step is to compute the address of the symbol string. This is done by adding st_name to STRTAB : const char *name = STRTAB + 0x120 == 0x320.

Note that SYMTAB access its entries as an array, therefore ELF32_sym should be aligned to 0x10 bytes. Now that we control st_name, we can basically force the resolver to relocate system and call system('sh') to a own the system :)

Writing the payload should be easy now that we have a clear image of the forged memory layout.

Ret-into-dl_resol x64

What is the difference between the x86 and x64 Elf32_Rel, Elf32_Sym structure instead of using a Elf64_Sym structure Elf64_Rela,.

The important thing here is that the change in the size of the structure.

- The size of the structure (8 byte) → Elf32_Rel Elf64_Rela structures, size (24-byte)
- The size of the Elf32_Sym structure (16 byte) → Elf64_Sym The size of the structure (24-byte)

Because of this reloc_offset value is an array of non-offset address Elf64_Rela structures should be an index.

```
typedef uint32_t Elf64_Word;
typedef uint64_t Elf64_Xword;
typedef int64_t  Elf64_Sxword;
typedef uint64_t Elf64_Addr;
typedef uint16_t Elf64_Section;

typedef struct
{
  Elf64_Addr    r_offset;               /* Address */
  Elf64_Xword   r_info;                 /* Relocation type and symbol index
*/
  Elf64_Sxword  r_addend;               /* Addend */
} Elf64_Rela;

typedef struct
{
  Elf64_Word    st_name;                /* Symbol name (string tbl index)
*/
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf64_Section st_shndx;               /* Section index */
  Elf64_Addr    st_value;               /* Symbol value */
  Elf64_Xword   st_size;                /* Symbol size */
} Elf64_Sym;
```

r_info

This member gives both the symbol table index, with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry will hold the symbol table index of the function being called. If the index is STN_UNDEF, the undefined symbol index, the relocation uses 0 as the symbol value. Relocation types are processor-specific. A relocation entry's relocation type or symbol table index is the result of applying ELF32_R_TYPE or ELF32_R_SYM, respectively, to the entry's r_info member:

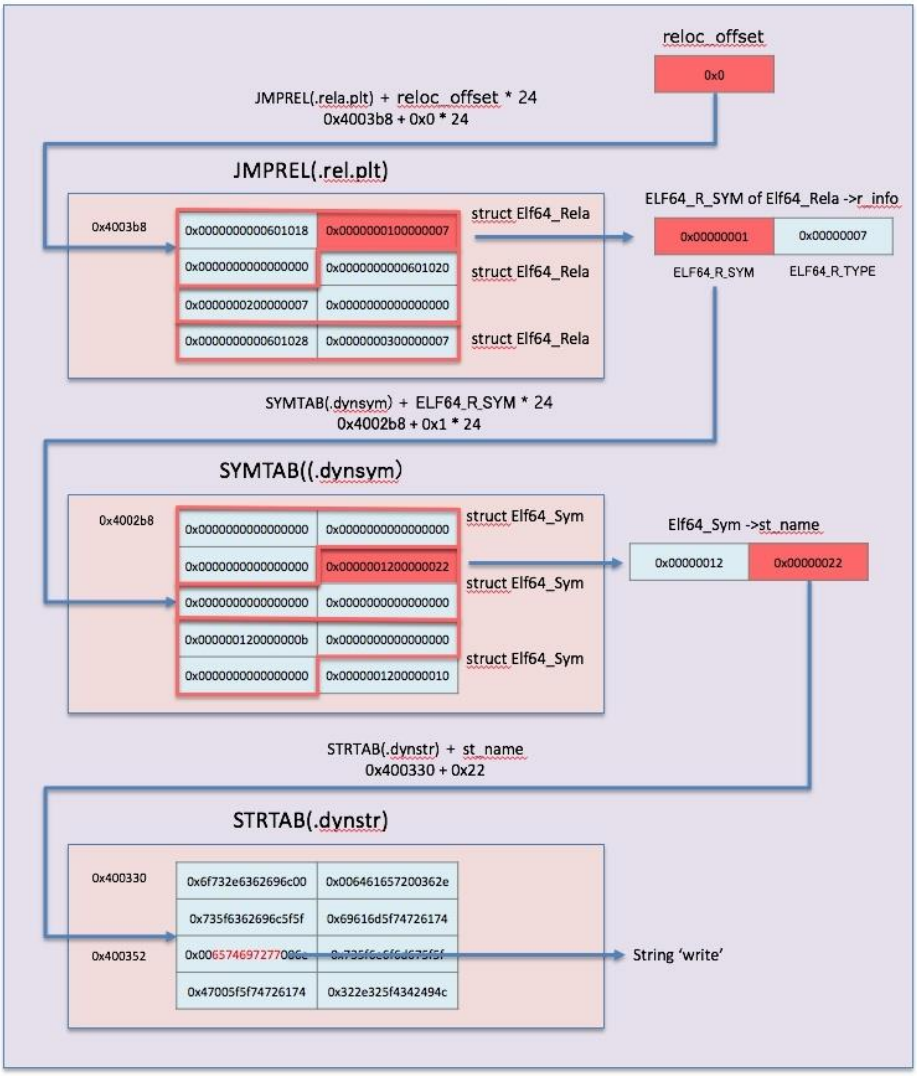
```
#define ELF32_R_SYM(info)             ((info)>>8)
#define ELF32_R_TYPE(info)            ((unsigned char)(info))
#define ELF32_R_INFO(sym, type)       (((sym)<<8)+(unsigned char)(type))

#define ELF64_R_SYM(info)             ((info)>>32)
#define ELF64_R_TYPE(info)            ((Elf64_Word)(info))
#define ELF64_R_INFO(sym, type)       (((Elf64_Xword)(sym)<<32)+ \
                                        (Elf64_Xword)(type))
```

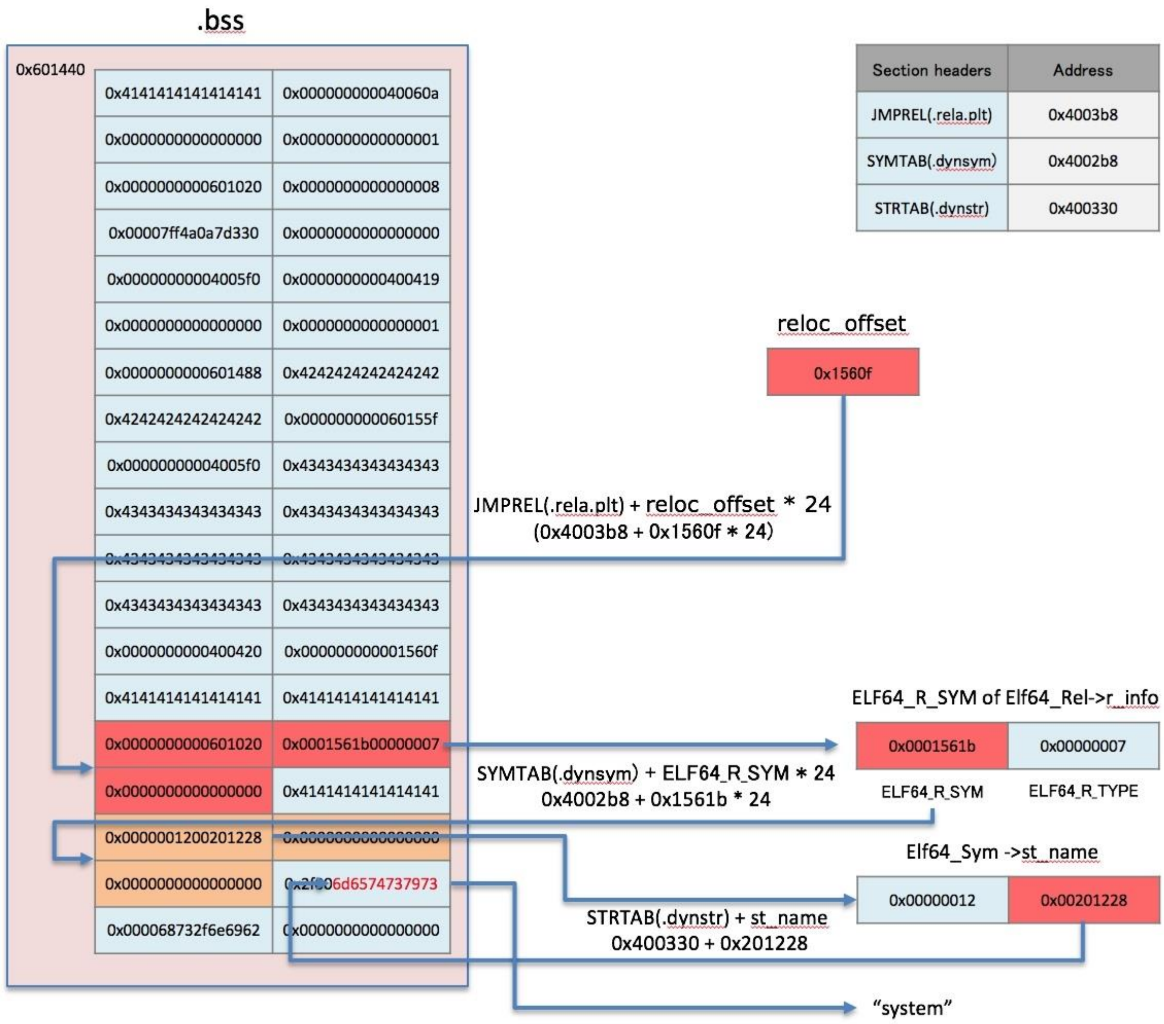
For Elf64_Rel and Elf64_Rela structures, the r_info field is further broken down into an 8-bit type identifier and a 24-bit type dependent data field:

```
#define ELF64_R_TYPE_DATA(info)       (((Elf64_Xword)(info)<<32)>>40)
#define ELF64_R_TYPE_ID(info)         (((Elf64_Xword)(info)<<56)>>56)
#define ELF64_R_TYPE_INFO(data, type) (((Elf64_Xword)(data)<<8)+ \
                                        (Elf64_Xword)(type))
```

_dl_fixup()



Section headers	Address
JMPREL(.rela.plt)	0x4003b8
SYMTAB(.dynsym)	0x4002b8
STRTAB(.dynstr)	0x400330



Return-to-csu

Command

```
objdump -M intel -d ./prog
```

```
4005f0:    4c 89 ea             mov    rdx,r13
4005f3:    4c 89 f6             mov    rsi,r14
4005f6:    44 89 ff             mov    edi,r15d
4005f9:    41 ff 14 dc          call   QWORD PTR [r12+rbx*8]
4005fd:    48 83 c3 01          add    rbx,0x1
400601:    48 39 eb             cmp    rbx,rbp
400604:    75 ea                jne    4005f0 <__libc_csu_init+0x40>
400606:    48 83 c4 08          add    rsp,0x8
40060a:    5b                   pop    rbx
40060b:    5d                   pop    rbp
40060c:    41 5c                pop    r12
40060e:    41 5d                pop    r13
400610:    41 5e                pop    r14
400612:    41 5f                pop    r15
400614:    c3                   ret
```

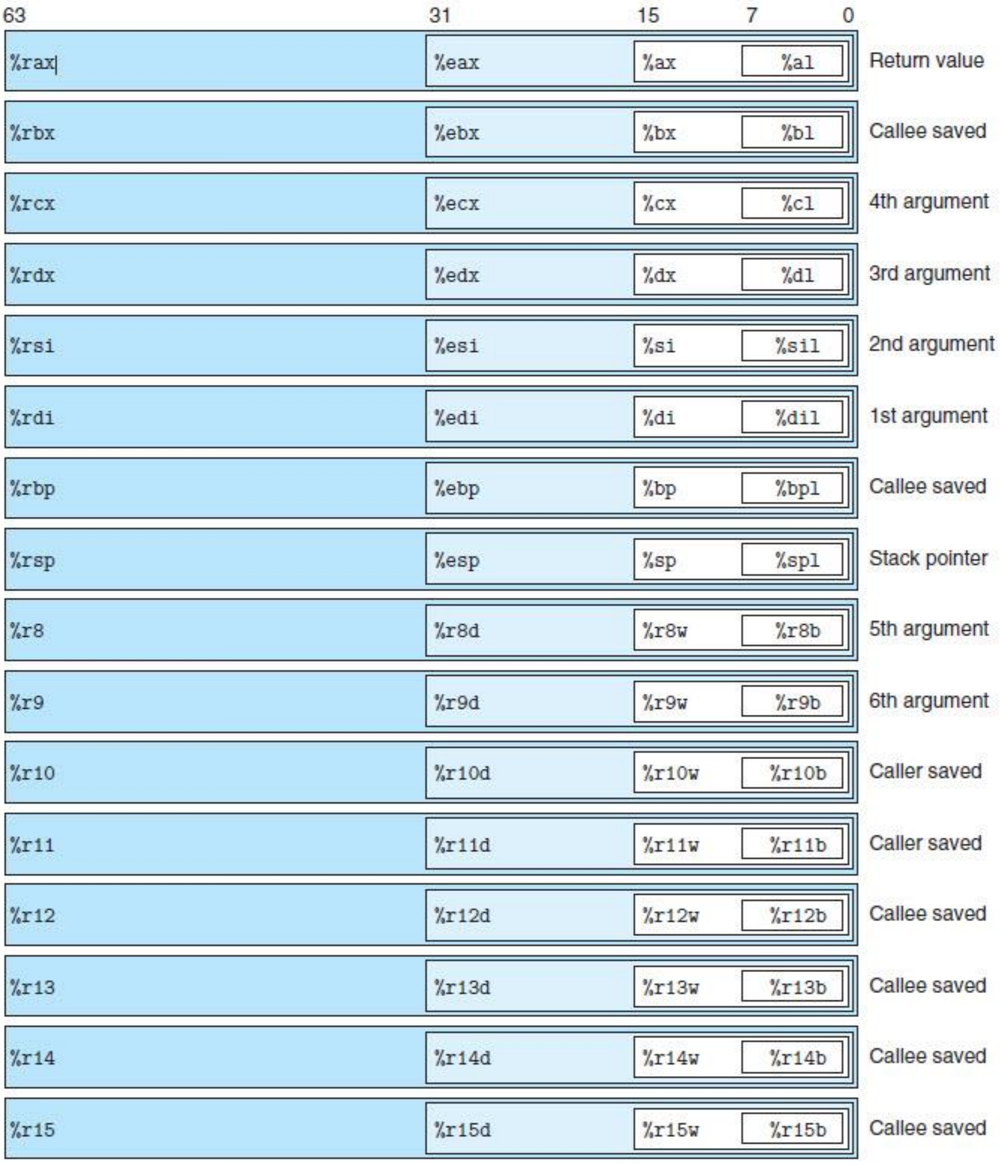
We can take advantage of the gadget of `__libc_csu_init`

- We can control rdx, rsi, edi with r13, r14, r15 register

The important thing is that in 64bit, the argument is saved in register.

Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Figure 3.28 Registers for passing function arguments. The registers are used in a specified order and named according to the argument sizes.



Reference:

1. *http://lia.deis.unibo.it/Courses/SicurezzaM1011/BufferoVerflow.pdf*
2. *Chapter 7: Linker CSAPP*
3. *https://gist.github.com/ricardo2197/8c7f6f5b8950ed6771c1cd3a116f7e62*
4. *http://phrack.org/issues/58/4.html?fbclid=IwAR3H6sjm3ouNN6rPlUuo2n1Rlu9JB-00SggAIP2fzokcGtT-hYOe0fj6jsQ*
5. *https://kileak.github.io/ctf/2018/0ctf-qual-babystack/*
6. *http://inaz2.hatenablog.com/entry/2014/07/15/023406*
7. *http://inaz2.hatenablog.com/entry/2014/07/27/205322*
8. *https://www.lazenca.net/pages/viewpage.action?pageId=19300744*