# 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))

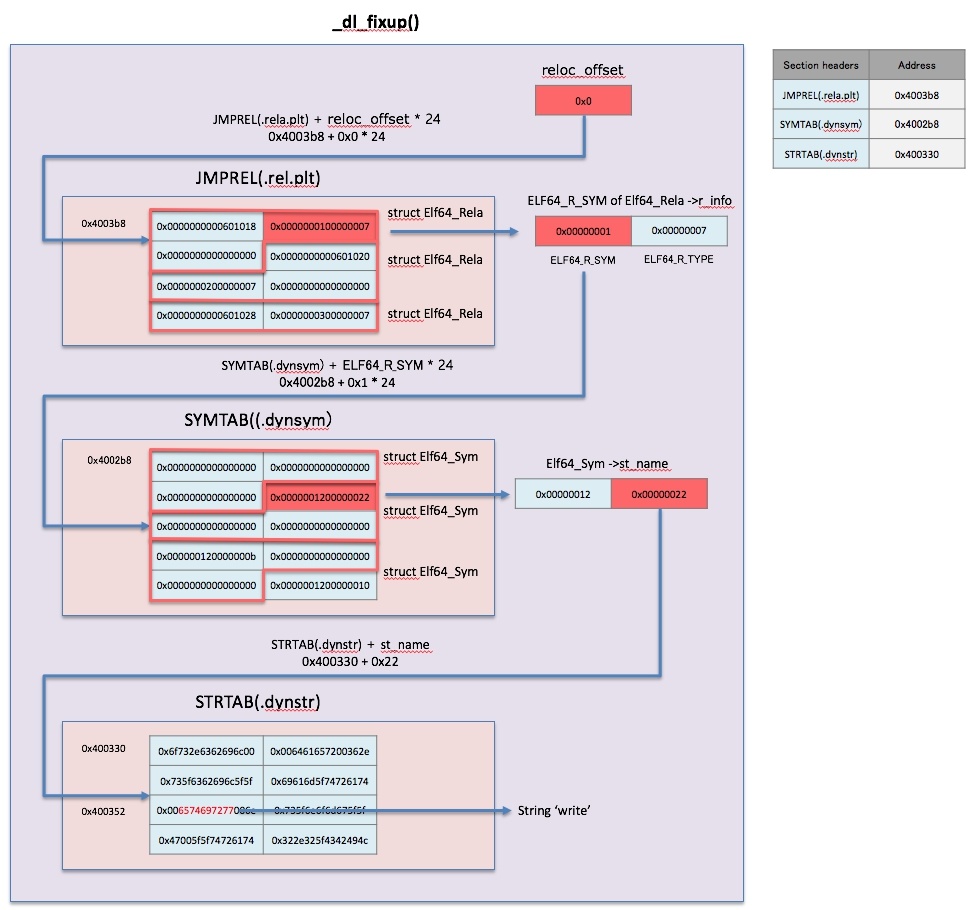
For Elf64\_Rel and Elf64\_Rela structures, the r\_info field is further broken down into an 8-bit type identifier and a 24-bit type dependent data field:

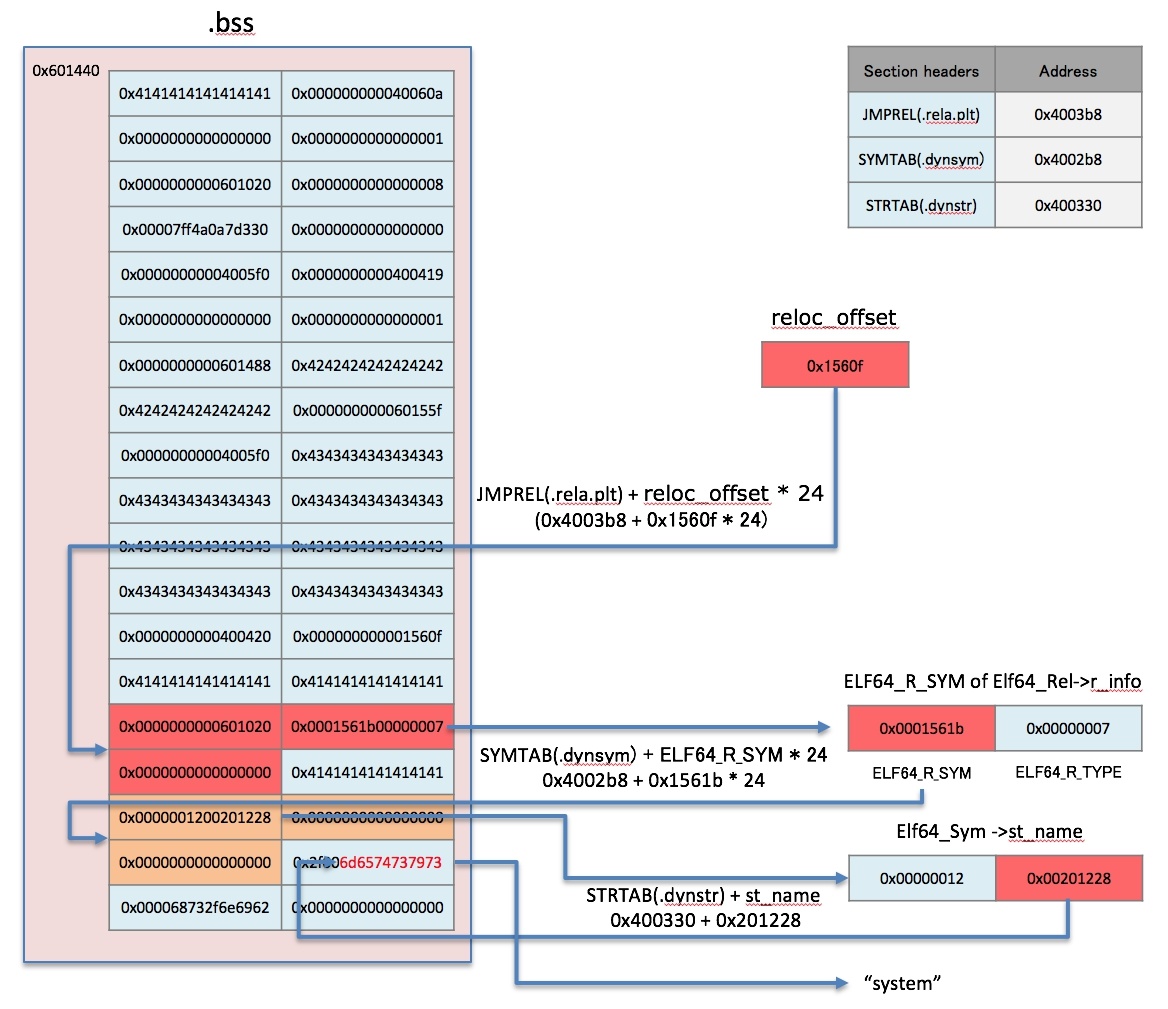
#define ELF64\_R\_TYPE\_DATA(info) (((Elf64\_Xword)(info)<<32)>>40)

#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))





# 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**

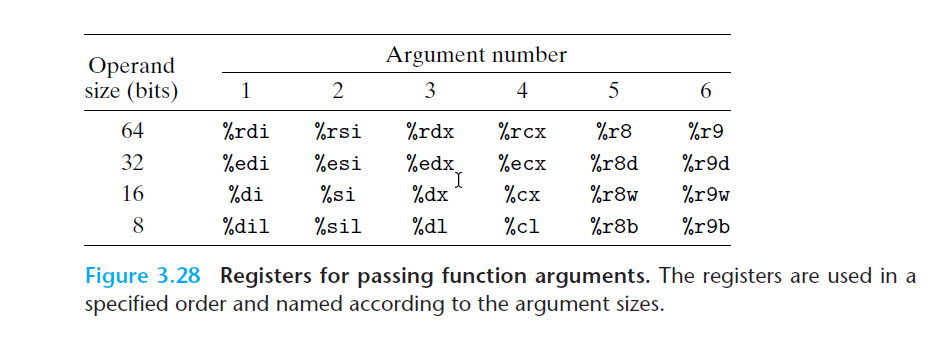
**400612:   41 5f                   pop    r15**

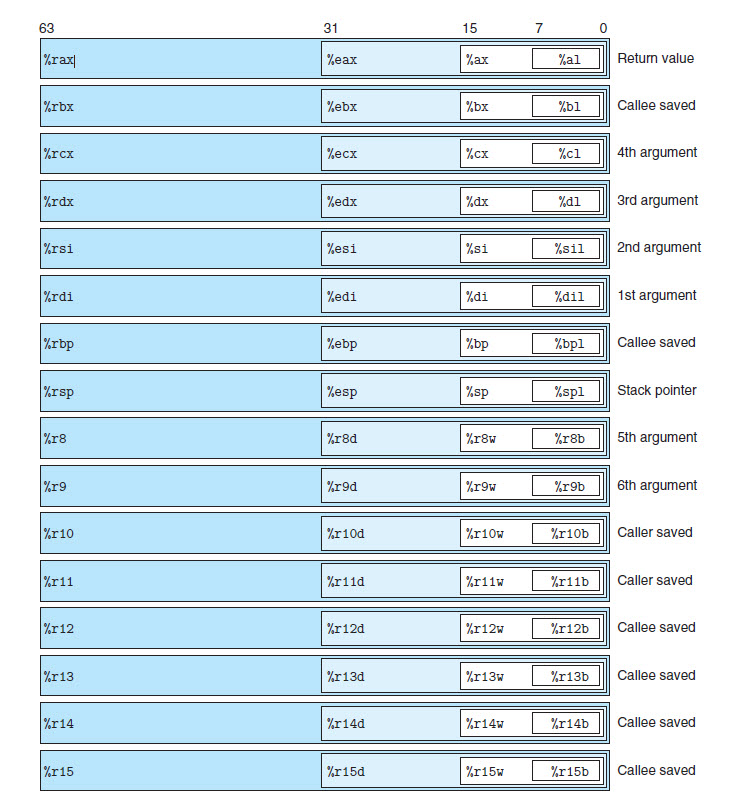
**400614:   c3                      ret**

 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.





*Reference:*

1. [*http://lia.deis.unibo.it/Courses/SicurezzaM1011/BufferoVerflow.pdf*](http://lia.deis.unibo.it/Courses/SicurezzaM1011/BufferoVerflow.pdf)
2. *Chapter 7: Linker CSAPP*
3. [*https://gist.github.com/ricardo2197/8c7f6f5b8950ed6771c1cd3a116f7e62*](https://gist.github.com/ricardo2197/8c7f6f5b8950ed6771c1cd3a116f7e62)
4. [*http://phrack.org/issues/58/4.html?fbclid=IwAR3H6sjm3ouNN6rPlUuo2n1RIu9JB-00SggAlP2fzokcGtT-hYOe0fj6jsQ*](http://phrack.org/issues/58/4.html?fbclid=IwAR3H6sjm3ouNN6rPlUuo2n1RIu9JB-00SggAlP2fzokcGtT-hYOe0fj6jsQ)
5. [*https://kileak.github.io/ctf/2018/0ctf-qual-babystack/*](https://kileak.github.io/ctf/2018/0ctf-qual-babystack/)
6. [*http://inaz2.hatenablog.com/entry/2014/07/15/023406*](http://inaz2.hatenablog.com/entry/2014/07/15/023406)
7. [*http://inaz2.hatenablog.com/entry/2014/07/27/205322*](http://inaz2.hatenablog.com/entry/2014/07/27/205322)
8. [*https://www.lazenca.net/pages/viewpage.action?pageId=19300744*](https://www.lazenca.net/pages/viewpage.action?pageId=19300744)