

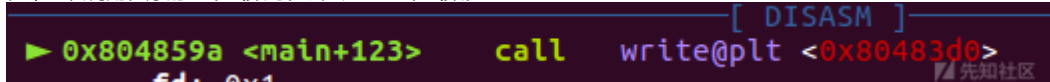
这几天学习了 return-to-dl-resolve 的知识 在这里做一些总结。
看了两个师傅的博客

[1nk3dHouse](#)
<http://pwn4.fun>

首先我们要了解到的是在ELF文件中 存在着一种延时绑定机制 叫做 : lazy binding。这种方式会在第一次调用一个函数时启动。

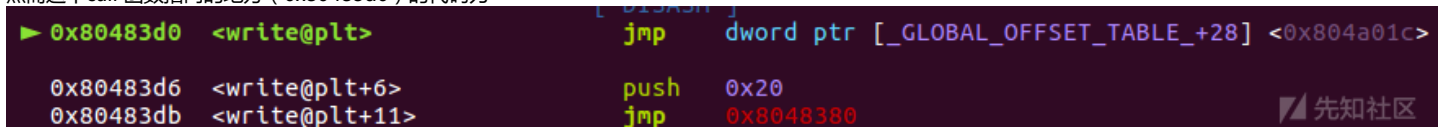
关于 lazy binding

1. 在第一次调用程序的一个函数时。会去call 一个函数。



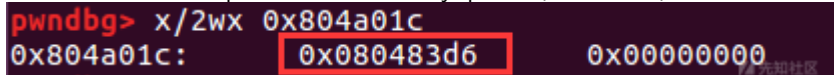
```
[ DISASM ]
> 0x804859a <main+123>  call  write@plt <0x80483d0>
    edi = 0x1
```

2. 然而这个call 函数指向的地方 (0x80483d0) 的代码为



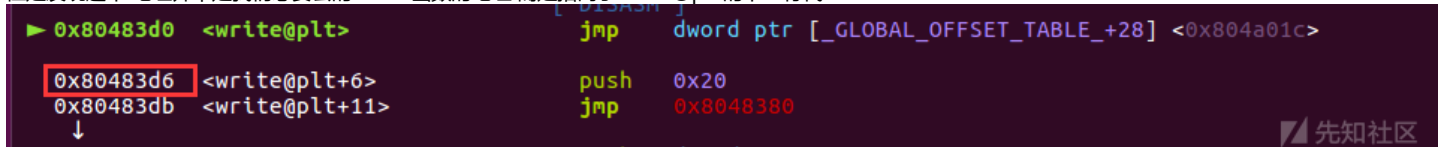
```
[ DISASM ]
> 0x80483d0 <write@plt>      jmp     dword ptr [_GLOBAL_OFFSET_TABLE_+28] <0x804a01c>
0x80483d6 <write@plt+6>      push    0x20
0x80483db <write@plt+11>     jmp     0x8048380
```

这个时候发现 在 write@plt 的第一个代码是一个 jmp 跳到 (0x804a01c) 保存的指针去。



```
pwndbg> x/2wx 0x804a01c
0x804a01c: 0x080483d6 0x00000000
```

但是发现这个 地址并不是我们想要去的 write 函数的地址 而是指向了 write@plt 的下一行代



```
[ DISASM ]
> 0x80483d0 <write@plt>      jmp     dword ptr [_GLOBAL_OFFSET_TABLE_+28] <0x804a01c>
0x80483d6 <write@plt+6>      push    0x20
0x80483db <write@plt+11>     jmp     0x8048380
↓
```

3. 接下来 程序进行了一个 push 和一个 jmp 指令。跳转到 (0x8048380) 而这个地址可以用 readelf -S 文件名来找到 发现这个地址是.plt的起始地址也就是PLT[0]

```
c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve$ readelf -S pwn1
There are 31 section headers, starting at offset 0x18a4:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	08048188	000188	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	080481ac	0001ac	00002c	04	A	5	0	4
[5]	.dynsym	DYNSYM	080481d8	0001d8	0000a0	10	A	6	1	4
[6]	.dynstr	STRTAB	08048278	000278	00006b	00	A	0	0	1
[7]	.gnu.version	VERSYM	080482e4	0002e4	000014	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	080482f8	0002f8	000020	00	A	6	1	4
[9]	.rel.dyn	REL	08048318	000318	000018	08	A	5	0	4
[10]	.rel.plt	REL	08048330	000330	000028	08	AI	5	24	4
[11]	.init	PROGBITS	08048358	000358	000023	00	AX	0	0	4
[12]	.plt	PROGBITS	08048380	000380	000060	04	AX	0	0	16
[13]	.plt.got	PROGBITS	080483e0	0003e0	000008	00	AX	0	0	8
[14]	.text	PROGBITS	080483f0	0003f0	000232	00	AX	0	0	16
[15]	.fini	PROGBITS	08048624	000624	000014	00	AX	0	0	4
[16]	.rodata	PROGBITS	08048638	000638	000008	00	A	0	0	4
[17]	.eh_frame_hdr	PROGBITS	08048640	000640	000034	00	A	0	0	4
[18]	.eh_frame	PROGBITS	08048674	000674	0000f4	00	A	0	0	4
[19]	.init_array	INIT_ARRAY	08049f08	000f08	000004	00	WA	0	0	4
[20]	.fini_array	FINI_ARRAY	08049f0c	000f0c	000004	00	WA	0	0	4
[21]	.jcr	PROGBITS	08049f10	000f10	000004	00	WA	0	0	4
[22]	.dynamic	DYNAMIC	08049f14	000f14	0000e8	08	WA	6	0	4
[23]	.got	PROGBITS	08049ffc	000ffc	000004	04	WA	0	0	4
[24]	.got.plt	PROGBITS	0804a000	001000	000020	04	WA	0	0	4
[25]	.data	PROGBITS	0804a020	001020	000008	00	WA	0	0	4
[26]	.bss	NOBITS	0804a040	001028	00000c	00	WA	0	0	32
[27]	.comment	PROGBITS	00000000	001028	000035	01	MS	0	0	1
[28]	.shstrtab	STRTAB	00000000	001799	00010a	00		0	0	1
[29]	.symtab	SYMTAB	00000000	001060	0004b0	10		30	47	4
[30]	.strtab	STRTAB	00000000	001510	000289	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
 0 (extra OS processing required) o (OS specific), p (processor specific)

4. 在jmp到这个地址后发现有一个push了一个值然后跳转到了 _dl_runtime_resolve 函数

```

0x8048380      push    dword ptr [_GLOBAL_OFFSET_TABLE_+4] <0x804a004>
0x8048386      jmp     dword ptr [0x804a008] <0xf7fee000>
↓
0xf7fee000 <_dl_runtime_resolve>      push    eax

```

然后调用这个函数实现延迟绑定。这个函数的原型为

`_dl_runtime_resolve(*link_map, rel_offset)` rel_offset 是 2. 中 push 的值, link_map 为 3. 中 push 的值。

从而执行 找到 write 函数的真实地址 从而在下次调用的时候直接跳转到 write 函数去。

结合 IDA 反编译

1. 第一次 call 函数会跳转到 .plt

```

.plt:080483D0      jmp     ds:off_804A01C
.plt:080483D0      _write

```

2. 然后跳转到 0x804a01c 的地址保存的地方 这个地方应该为

```
.got.plt:0804A01C off_804A01C      dd offset write
```

3. 但是第一次 还没有保存write 函数真实的 地址所以会跳转到 .plt 的下一行代码0

```
.plt:080483D0          jmp     ds:off_804A01C
.plt:080483D0 _write      endp
.plt:080483D0
.plt:080483D6 ; -----
.plt:080483D6          push    20h
.plt:080483DB          jmp     sub_8048380
```

4. 从而去 push 参数 实现 将 write函数的真实地址保存在 .got.plt（保存函数引用位置） 上方便下次调用这个函数时能直接找到 write函数的地址

上面的就是延迟绑定的基本原理为了更深入了解return-to-dl-resolve 的实现我们还要学习一下 elf 文件。

ELF 可执行文件 由 ELF头部，程序头部表和对应的段，节头部表和其对应的节组成。

- 在程序头部表中会有一个 PT_DYNAMIC 的段，这个段包含 .dynamic 节区，这个类型的段的 结构为（这个 .dynamic 节区也被称为“重定位节区”可以用到 readelf -d 查看）：

```
typedef struct {
    Elf32_Sword d_tag;
    union {
        Elf32_Word d_val;
        Elf32_Addr d_ptr;
    } d_un;
} Elf32_Dyn;
```

readelf -d

c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve\$ readelf -d pwn1

Dynamic section at offset 0xf14 contains 24 entries:

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000c	(INIT)	0x8048358
0x0000000d	(FINI)	0x8048624
0x00000019	(INIT_ARRAY)	0x8049f08
0x0000001b	(INIT_ARRAYSZ)	4 (bytes)
0x0000001a	(FINI_ARRAY)	0x8049f0c
0x0000001c	(FINI_ARRAYSZ)	4 (bytes)
0x6ffffef5	(GNU_HASH)	0x80481ac
0x00000005	(STRTAB)	0x8048278
0x00000006	(SYMTAB)	0x80481d8
0x0000000a	(STRSZ)	107 (bytes)
0x0000000b	(SYMENT)	16 (bytes)
0x00000015	(DEBUG)	0x0
0x00000003	(PLTGOT)	0x804a000
0x00000002	(PLTRELSZ)	40 (bytes)
0x00000014	(PLTREL)	REL
0x00000017	(JMPREL)	0x8048330
0x00000011	(REL)	0x8048318
0x00000012	(RELSZ)	24 (bytes)
0x00000013	(RELENT)	8 (bytes)
0x6fffffff	(VERNEED)	0x80482f8
0x6fffffff	(VERNEEDNUM)	1
0x6fffffff0	(VERSYM)	0x80482e4
0x00000000	(NULL)	0x0

- 在节区中包含着 目标文件的 所有的信息用一个结构保存着。(其中Type为REL的节区包含重定位表项。可以用 readelf -S 查看)

```
typedef struct {
    Elf32_Word sh_name;      // 节头部字符串表节区的索引
    Elf32_Word sh_type;      // 节类型
    Elf32_Word sh_flags;     // 节标志, 用于描述属性
    Elf32_Addr sh_addr;      // 节的内存映像
    Elf32_Off  sh_offset;    // 节的文件偏移
    Elf32_Word sh_size;      // 节的长度
    Elf32_Word sh_link;      // 节头部表索引链接
    Elf32_Word sh_info;      // 附加信息
    Elf32_Word sh_addralign; // 节对齐约束
    Elf32_Word sh_entsize;   // 固定大小的节表项的长度
} Elf32_Shdr;
```

readelf -S

```
c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve$ readelf -S pwn1
There are 31 section headers, starting at offset 0x18a4:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	08048188	000188	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	080481ac	0001ac	00002c	04	A	5	0	4
[5]	.dynsym	DYNAMIC	080481d8	0001d8	0000a0	10	A	6	1	4
[6]	.dynstr	STRTAB	08048278	000278	00006b	00	A	0	0	1
[7]	.gnu.version	VERSYM	080482e4	0002e4	000014	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	080482f8	0002f8	000020	00	A	6	1	4
[9]	.rel.dyn	REL	08048318	000318	000018	08	A	5	0	4
[10]	.rel.plt	REL	08048330	000330	000028	08	AI	5	24	4
[11]	.init	PROGBITS	08048358	000358	000023	00	AX	0	0	4
[12]	.plt	PROGBITS	08048380	000380	000060	04	AX	0	0	16
[13]	.plt.got	PROGBITS	080483e0	0003e0	000008	00	AX	0	0	8
[14]	.text	PROGBITS	080483f0	0003f0	000232	00	AX	0	0	16
[15]	.fini	PROGBITS	08048624	000624	000014	00	AX	0	0	4
[16]	.rodata	PROGBITS	08048638	000638	000008	00	A	0	0	4
[17]	.eh_frame_hdr	PROGBITS	08048640	000640	000034	00	A	0	0	4
[18]	.eh_frame	PROGBITS	08048674	000674	0000f4	00	A	0	0	4
[19]	.init_array	INIT_ARRAY	08049f08	000f08	000004	00	WA	0	0	4
[20]	.fini_array	FINI_ARRAY	08049f0c	000f0c	000004	00	WA	0	0	4
[21]	.jcr	PROGBITS	08049f10	000f10	000004	00	WA	0	0	4
[22]	.dynamic	DYNAMIC	08049f14	000f14	0000e8	08	WA	6	0	4
[23]	.got	PROGBITS	08049ffc	000ffc	000004	04	WA	0	0	4
[24]	.got.plt	PROGBITS	0804a000	001000	000020	04	WA	0	0	4
[25]	.data	PROGBITS	0804a020	001020	000008	00	WA	0	0	4
[26]	.bss	NOBITS	0804a040	001028	00000c	00	WA	0	0	32
[27]	.comment	PROGBITS	00000000	001028	000035	01	MS	0	0	1
[28]	.shstrtab	STRTAB	00000000	001799	00010a	00		0	0	1
[29]	.symtab	SYMTAB	00000000	001060	0004b0	10		30	47	4
[30]	.strtab	STRTAB	00000000	001510	000289	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
 0 (extra OS processing required) o (OS specific), p (processor specific)

- 记录重定位信息的节区中。 .rel.plt 用于函数定位 .rel.dyn 用于变量定位。而 REL 类型的节 的结构体为 (可以用 readelf -r 查看着两个节区) 注意在 REL 类型的节中 .rel.plt 中的 info 应该如0x607 最后一位为7 而这个7 是为了在 之后找到 对应的 R_386_JUMP_SLOT。


```
typedef struct {
    Elf32_Addr r_offset;    // 对于可执行文件，此值为虚拟地址
    Elf32_Word r_info;      // 符号表索引
} Elf32_Rel;

#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))
```

readelf -r

c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve\$ readelf -r pwn1

```
Relocation section '.rel.dyn' at offset 0x318 contains 3 entries:
  Offset      Info      Type           Sym.Value    Sym. Name
08049ffc      00000306 R_386_GLOB_DAT 00000000     __gmon_start__
0804a040      00000905 R_386_COPY     0804a040     stdin@GLIBC_2.0
0804a044      00000705 R_386_COPY     0804a044     stdout@GLIBC_2.0

Relocation section '.rel.plt' at offset 0x330 contains 5 entries:
  Offset      Info      Type           Sym.Value    Sym. Name
0804a00c      00000107 R_386_JUMP_SLOT 00000000     setbuf@GLIBC_2.0
0804a010      00000207 R_386_JUMP_SLOT 00000000     read@GLIBC_2.0
0804a014      00000407 R_386_JUMP_SLOT 00000000     strlen@GLIBC_2.0
0804a018      00000507 R_386_JUMP_SLOT 00000000     __libc_start_main@GLIBC_2.0
0804a01c      00000607 R_386_JUMP_SLOT 00000000     write@GLIBC_2.0
```

其中的 .rel.plt 就是 IDA 反编译过后 .got.plt 的值

- 查看资料发现 .got 节保存着全局变量偏移表，.got.plt 节保存全局函数偏移表。也就是 REL 结构体中的

```
typedef struct {
    Elf32_Addr r_offset;    // 对于可执行文件，此值为虚拟地址
    Elf32_Word r_info;      // 符号表索引
} Elf32_Rel;

#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))
```

在我们实现 延迟绑定时 还会调用到一个类型为 SYM 的节叫做 .dynsym，这个节 的结构为（可以用到 readelf -s 查看）主要用于找到 REL 对应的函数符号表信息。用到 ELF32_R_SYM(Elf32_Rel->r_info)

```
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_Word st_shndx;     // Section index
} Elf32_Sym;
```

readelf -s

```
c4m3l@c4m3l-virtual-machine:~/Desktop/practise/re2dlresolve$ readelf -s pwn1
```

Symbol table '.dynsym' contains 10 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	setbuf@GLIBC_2.0 (2)
2:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	read@GLIBC_2.0 (2)
3:	00000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
4:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	strlen@GLIBC_2.0 (2)
5:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.0 (2)
6:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	write@GLIBC_2.0 (2)
7:	0804a044	4	OBJECT	GLOBAL	DEFAULT	26	stdout@GLIBC_2.0 (2)
8:	0804863c	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used
9:	0804a040	4	OBJECT	GLOBAL	DEFAULT	26	stdin@GLIBC_2.0 (2)

先知社区

然后上一个 .dynsym 我们得到了 num 然后通过这个 num 在 .dynstr 中找到对应的函数的字符串。Elf32_Sym[num]->st_name=0x4c█.dynsym + Elf32_Sym_size█0x10█ * num) 在.dynsym的地址基础上加上 num*0x10 这个地址保存的值，这个值是我们需要的 函数字符串保存在 .dynstr节上的偏移。比如 这个write 函数的 .rel结构体中 info = 0x607 所以这个num = 6 <-- (0x607>>8)

通过偏移找到write函数的字符串在 .dynstr中的偏移

用.dynstr基地址+偏移得到函数字符串

总结return-to-di-resolve调用步骤：

[illegible]

这知识程序的运行方法 想要利用这个 漏洞要 充分理解明白这个 机制的运行。

2. 通过 `pym` 找到对应函数的字符串从而找到函数的真实地址。

修改.dynstr节中的字符串为我们需要的字符串这样就能在绑定是找到我们想利用的函数。

点击收藏 | 2 关注 | 2

[上一篇：某shop API接口前台注入\(通...](#)
[下一篇：Rabin加密算法和n次同余方程](#)

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](#)