**Pwn学习笔记**

1. **前言**

即将踏入研究生生活，由于专业方向是网安，因此终归是要在各个方向中选一条。目前暂定为假期先学pwn方向，等这个方向学完之后再决定是否深入二进制。同时考虑在之后的空闲时间学学web。

目前打算先跟着CTF-wiki走，按照pwn里面的顺序学习。在刚学没两天就已经遇到很多的问题。由于不久后还要换电脑，为了避免问题反复重复，于是决定破天荒的开始写日志，把遇到的问题和解决都记录下来。

1. **学习进度**

2022-6-29：看完了X86的栈介绍和溢出原理，由于未配置好环境，实验部分暂时没有重复。

2022-6-30：基本ROP看了ret2text并且完成了重复实验，ret2shellcode截止到写出这一条，还没有完成实验复现。正式今天遇到的不少问题促使我开始写日志记录。

2022-7-2：昨日出去游玩一天，今天继续。发现ASLR每次虚拟机开机都会重置为2，记得要改回0。今天终于整完了ret2shellcode和ret2syscall。准备学习ret2libc。明天先看文章理解GOT和PLT（https://baijiahao.baidu.com/s?id=1663915740492408592）

2022-7-3：开组会，所以没搞太多。主要弄明白了GOT和PLT。同时为了搞明白为什么glibc的system需要在栈里先设置返回值地址和指令字符串时，顺带看了一篇讲解system的文章，其主要内容是分析了为什么当缓冲区溢出时字符串太长会导致调用system函数失败。其主要原因是栈溢出太长导致覆盖了环境变量\_\_environ。

（https://wenku.baidu.com/view/62ba0468bd1e650e52ea551810a6f524ccbfcbb1.html）

今天最后还是把ret2libc的三个例题做完了，很爽，明天做作业题。

2022-7-4：四个作业题写完了，收获了很多，见了很多奇妙的操作。提一个比较重要的，gbd attach后用continue继续调试，用run就重启了。

2022-7-9：咕了很久，主要是5号买了新电脑，配环境先配了两天。7号想整活换个用户名，差点把系统搞崩了，得亏留了一手提前注册了管理员账户。8号学习了理论部分，今天才开始实践。不得不说中级的确实挺难

2022-7-12:10号出门，到了高级ROP后，为了前置知识昨天花了一天时间再看ctf-wiki里execute的elf文件内容，看的云里雾里，很多符号因为简称所以不太记得住，但对整体还是有个了大概印象，并且加上今天的学习大致弄懂了动态链接的过程。

2022-7-14：昨天忘了写了，今天一块补上。主要就是开始实践高级ROP的内容了，昨天做了no relro，内容是如何手动触发函数解析以及替换掉解析的函数，方法就是伪造字符串表后改.dynamic就行。今天则是partial relro，不能再改.dynamic了，然后逐渐深入伪造.rel.plt, .dynsym。后面应该还有.dynstr，但今天就到此为止了。卡在stage4的一个没看明白的毛病上，明天再看。

2022-7-15：大致弄懂了stage4的偏移逻辑，明天开始搞stage5

2022-7-19：前几天忘了写进度日志，但还是做了东西。主要就是已经完成了所有的stage，现在开始看X64的情况。

2022-7-21：昨天摸了，今天主要是看了SROP前的部分和SROP的基本原理介绍，同时通过实验得知一个具有冲击性的事实。我在win11上用visual studio生成的exe，栈居然是从低地址往高地址增长的。理论上这样栈就溢出不了，先往后学，可能有我不知道的事情。

2022-7-23：昨天把SROP给复现了一遍，不是很难。今天大概扫了一下最后的花式栈溢出，主要是遇到一些情况时采取的策略，并给了一些简单例题。只是量比较多，而且很多不是决定性的技巧，而是一些小trick，所以就不打算看完。为了赶进度，明天开始直接进入Format String章节

2022-7-26：前两天把format string的介绍和利用搞完了，今天做了例子的两个题，遇到一点问题。

2022-7-30：时间突然就飞逝了，主要是太摸了，今天看了两个题，没实操，但弄懂了主要思路。

2022-8-2：昨天和前天把堆的前置知识看了个七七八八，但在后面有很多带代码的具体实现没怎么看。主要是了解了相关数据结构，知道了大致的分配和回收特性。今天把堆溢出的基础知识看了。

2022-8-4：主要看了堆中off-by-one的基础原理，给了一个例题没能复现出来，测试发现指向description的前提无法实现。同时看了一下后面的部分解析，发现有些原理没学，打算先跳过这一节，往后学。

2022-8-8：看完了chunk extend and overlapping

2022-8-9：看了第一个例题，未能复现，但思路可叙述

2022-8-10：往下继续看例题，但仍然未能复现，unlink利用依然失败。跳过两个例题，开始看use after free

2022-8-12：昨天实践了了use after free的例题。今天看了fastbin attack的一个例题和相关的常见利用方式。跳了两个例题，顺带看了一下unsorted bin attack的相关原理。

2022-8-16：前几天看完了unsorted bin attack，large bin attack。今天基本看完了tcache attack的原理。明天开始看相关例题。

2022-8-17：今天做了例题 LCTF2018 PWN easy\_heap，顺带把昨天没看的libc leak看了。题目挺难得，我的总结比较抽象，下次复现的时候最好画图。之后短时间内可能不会看pwn了，因为实验室那边安排了任务，之后的重心会放到reverse上面去。

2022-9-7：即将40分钟大分享，打算讲讲pwn。因此这段时间顺带复习了一下之前学习的内容。同时发现了一个不错的pwn总结网址：https://www.wangan.com/docs/551

1. **环境配置（待更新和添加）**

Windows10下的软件有：IDAPro（包括X86和X64），VMMap，VMware，visual C++，python

Ubuntu22.04-desktop（虚拟机）下的软件有：idafree77\_linux（仅X64，可在官网下），python3.10.4（pwntools），gdb（peda；动态调试器，pwngdb），checksec，ropgadget，LibcSearcher，

1. **问题，解决，实验记录与总结**
   1. **stack overflow**
      1. **X86**
         1. **栈介绍**

没什么好说的介绍栈的基本知识，里面两个链接讲的很详细，注意要看。最重要的是固定的相对位置，这里用文字描述一下，从左到右为从高地址到低地址。注意栈是往低地址增长的。左边的是调用函数的函数栈帧

...->EBP->本地变量->调用函数的参数（从n到1，即1靠近返回地址）->返回地址|EBP->本地变量（从1到n，即1靠近返回地址）->下一个调用函数的参数->返回地址->...

* + - 1. **栈溢出原理**

开始进入正片。

栈溢出的基本前提：程序必须往栈上写数据。写入的数据大小没有被良好控制。

下面给出了一个基本示例，内容很好理解。给了一个容易被越界的字符数组，然后又通过不控制输入长度的gets函数获取输入。只需要通过IDA知道攻击函数的地址，然后输入时控制输入数据，使得攻击函数地址正好覆盖到返回地址即可。但有以下几个要注意的

* 1. 攻击函数地址所在段要有执行权限。
  2. 不开启堆栈溢出保护（ -fno-stack-protector）
  3. 关闭PIE避免加载基址被打乱（-no-pie）
  4. Linux里有地址空间分布随机化（ASLR）的机制要关掉 (/proc/sys/kernel/randomize\_va\_space)。

本例中，buff到返回地址的长度为buff长度+4（EBP）。故这个长度内容随便填充，然后写入攻击地址即可。

写入时需要注意的：

1. 由于直接输入写不了16进制数，所以需要用到pwntools的p32将地址转换一下。默认是小端序，和实际是一样的。
2. Pwntools在ubuntu上用比较好
   * + 1. **基本ROP**

前面叙述的原理比较复杂，写下这段话时还没搞懂。先记录后面的，待日后修改。

* 1. **Ret2text**

和之前那个本质是一样的，通过控制程序执行程序本身已有的的代码 (.text)来攻击。实际上也可以控制执行好几段不相邻的代码（gadgets）。给出的例子中就是直接跳转到打开shell的代码。

这里也介绍了攻击的基本流程。

1. 通过checksec检查目标程序的状态，例如是多少位程序，受到什么保护，有什么限制等。
2. 通过IDA查看源码，如果是32位程序，ubuntu上不能反编译，所以最好用windows的IDA查看源码，反编译后看起来更直观，直接看汇编有点看天书。
3. （补充）打开string，从这其中观察有没有可能的漏洞，例如本例中有/bin/sh。
4. 寻找可供跳转（攻击）的代码段，本例中就是在secure函数（一点都不安全）里找到了打开shell的代码（事后发现，实际上是通过string里观察到/bin/sh发现的），进而通过覆盖ret跳转来获得了系统的shell。
5. 构造payload。需要观察buf与ret之间的距离来计算中间填充多少。例子中给了个比较聪明的办法。根据观察，在gets之前esp跳转到esp+1Ch。文中认为这体现了buf首地址与esp的距离。然后在get（注意从计算距离后道这里的过程，esp是不变的）处下个断点查看esp和ebp的距离（看来ebp后是return是亘古不变的）。拿后者减前者就是两者的距离，+4就是return了。我个人初期采取的办法比较无脑，输个有代表性的123456（ascii比较显眼，313233343536），然后在gets后从内存里找一找，一定就在ebp附近。然后再调试到ret前，看看esp的地址（当时并不是那么确定ebp和ret的位置关系，毕竟那可是main啊）。然后得出结果。
6. Python写个程序A进去就完事了
   1. **Ret2shellcode**

折磨人的教学。这里我学了几乎一天。先是因为32位程序在ubuntu里不能通过IDA64反编译。最后痛下决心开始双平台作战。然后是文中说的“调试程序”，我在这里卡了很久，因为我完全不知道他是怎么调试的，直到后来我通过其调试内容搜索关键词，才知道了这个调试用的是gdb（peda）。

当时研究进入了死锁。因为如果在ubuntu里确实可以运行程序，因为那玩意本来就是linux下课运行的。但是文中调试时使用了vmmap来查看bss段是否可执行（因为有个buf2的变量，想在里面写shellcode来执行程序，checksec时明确表示了有RWX段，即可读可写可运行）。百度一下，发现这玩意是在windows下用的，ubuntu下好像没有。转而去整windows。

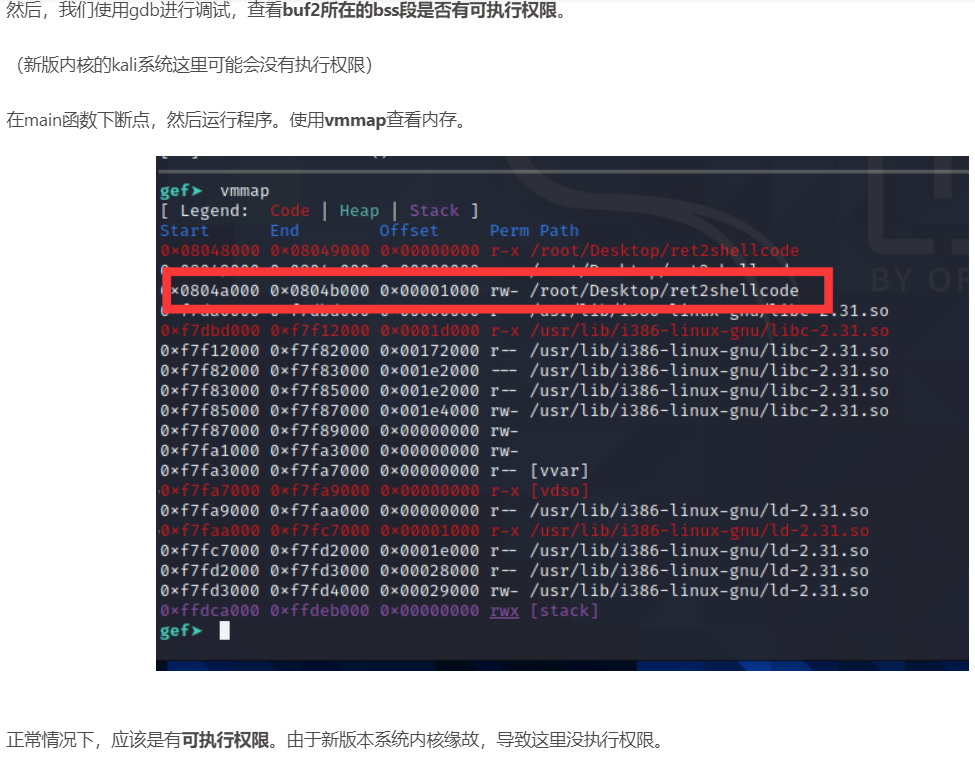
在windows下，IDA默认的windos调试器不能用了，估计因为那就不是windows的程序。没办法下了一个bochs调试器（其实中途还尝试过连接虚拟机的linux调试器，结果发现不是我理解的那个意思。意思是远程调试linux的程序，遂作罢）。配了半天调试器能用了开始调试，结果发现调试后vmmap找不到这个进程，尽管调试时已经明确给出进程号，但就是死活找不到。然后就寄了。

最后经过种种机缘巧合，才在网上偶然发现教程里那个调试用的应该是gdb，在完笔之前刚下完。之后测试一下能不能复现操作。

确实可以复现操作，现象完全一致，唯一的问题是我的vmmap告诉我，bss段没有执行权限。我现在去看看为什么。

花了很长时间，最后在一篇博客里得出结论，是内核问题，真是草了。

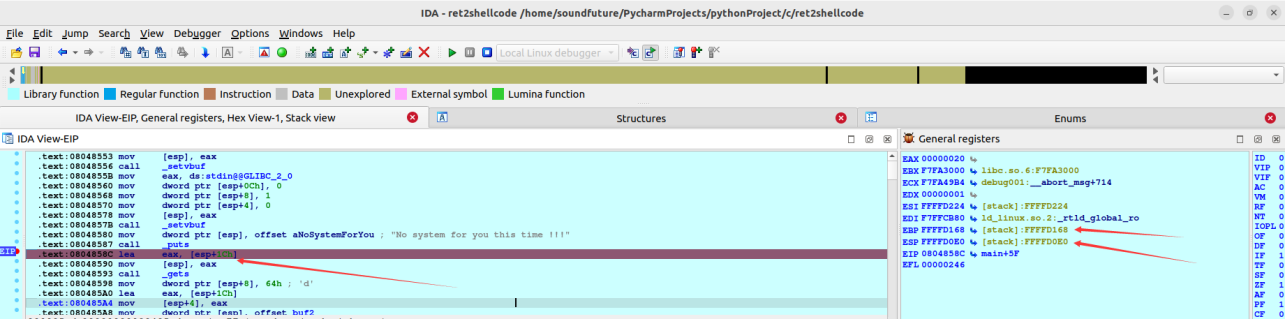
(https://blog.xxxb.cn/2021/05/16/BasicROP-ret2ShellCode/)

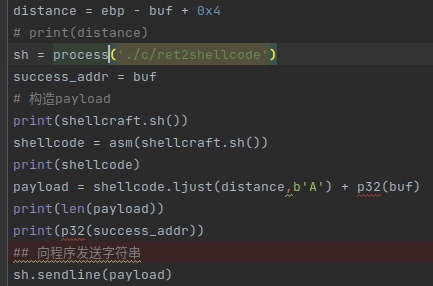


另一篇博客提到，可以通过mprotect()函数改写赋予bss执行权限，明天再来尝试。

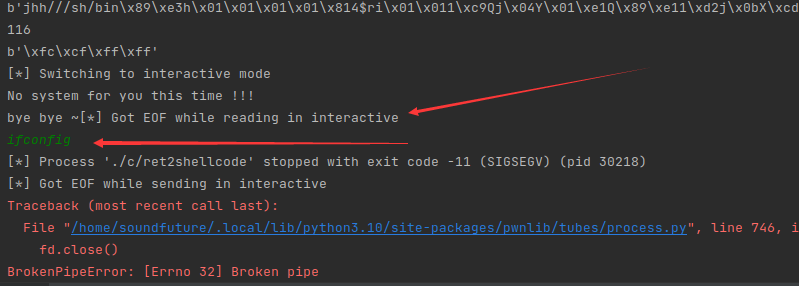
（https://cloud.tencent.com/developer/article/1740266）

**2022-7-2更新：**

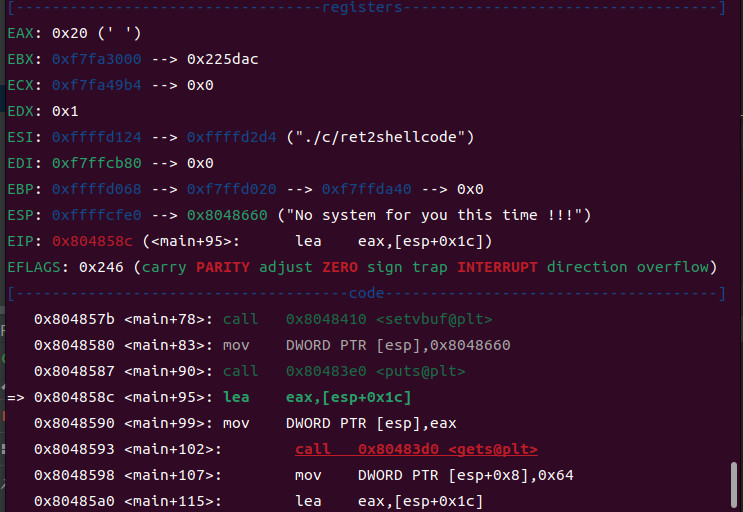
注意到栈内是有执行权限的，所以打算先把shellcode写在栈里面执行。但这却是噩梦的开端。  
 初期实践起来很简单，观察IDA，可以看出buf和ebp的地址分别是0xFFFFD0FC，0xFFFFD168。所以ret地址为0xFFFFD16C。使用pwntools的asm(IMG_260shecllcraft.sh)构造拿shell的code，然后剩下的随便填充就行。  




然而程序执行后出现了问题



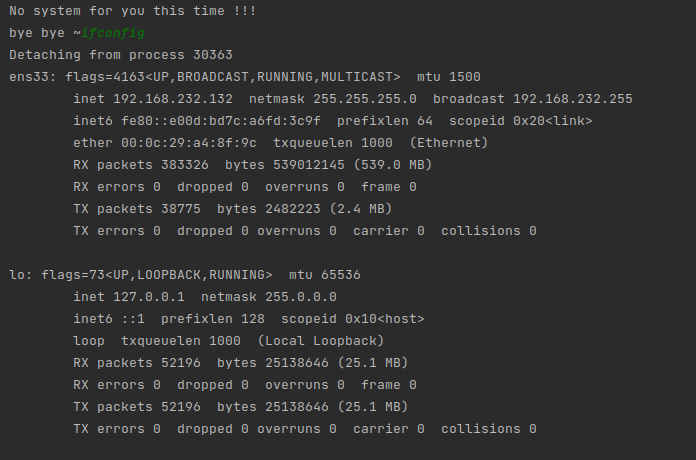
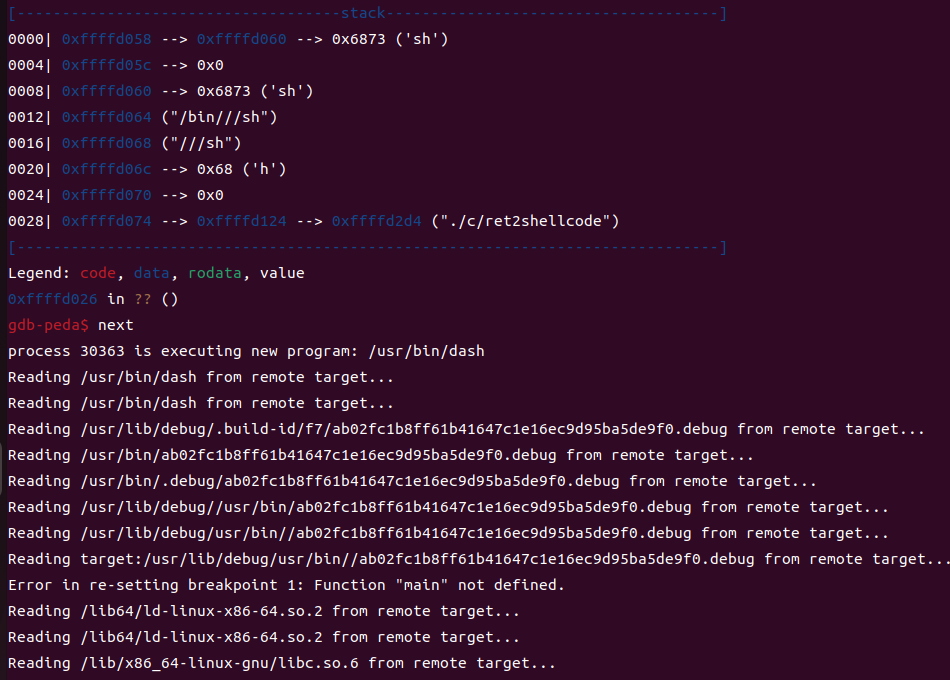
注意到有个提示语Got EOF while reading in interactive，同时我再输入ifconfig时程序直接报错退出了。  
我查了一下这个提示语，有很多解释和我比较接近，但没有解决我的问题。  
然后我觉得用gdb.debug单步调试一下看看，然后整个事情最神奇的地方出现了，也是我最不能理解的地方，下图显示了寄存器和代码的地址和内容。



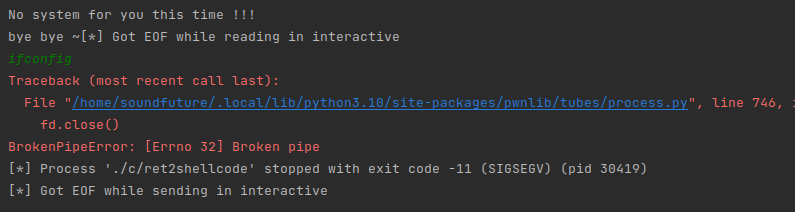
注意到我在执行到和上述IDA同一个代码时，EBP和ESP都很奇怪的比IDA中的地址要少0x100。但是汇编代码的地址却是对得上的。（事后解释是IDA可能在调试时在底部添加了部分环境变量导致错位）  
 于是我在python代码里根据gdb显示的地址修正了buf和ebp的地址，如下，就是比原来的地址要少0x100



此时再次通过gdb.debug单步调试我的代码，发现他顺利的跳转到我的buf首地址并执行了shellcode，同时我在控制台输入ifconfig后，控制台跳出了相关信息。



正当我欣喜若狂，觉得事情结束的时候，决定最后有仪式感的把gdb.deug改回process，让他自己完整跑一遍。结果如下

IMG_268

它又报错了，我傻了。

最终问题解决了，可能的解释是用IDA和gdb.debug都会在栈底创建环境变量。虽然数据显示有点不太对的上这个解释。



上图中，可以看到，IDA的buf位置是栈里最低的，因为地址最大。其次是用GDB的attach时buf的地址，最后是GDB.debug得到的buf地址。所以更可能的解释是GDB自己会创建变量，IDA不会，而程序在python里用process运行时也会创建少许变量，attach是后来附加的，理当最接近真实环境，所以attach最好。不过总之，以后还是gdb.attach来看栈内变量的地址最为准确。并且最好不要把shellcode写在栈里面，太麻烦了。

用mprotect改写bss有点没看明白。下次在搞。先做下一题。

* 1. **sniperoj-pwn100-shellcode-x86-64**

不是很难，比起常规的ret2shellcode，这里开启了PIE保护，栈内可以执行。但是程序里直接把buf地址给打印出来了。因此学会了新操作，就是recvuntil，这个方法可以把程序输出的结果给截取出需要的部分。此处只要截取出buf地址然后再构造payload就可以了。唯一要注意的是，此处buf距离RBP（是X64程序）仅0x10的距离，加上RBP本身也只有0X18的空间构造。太小了。所以干脆在ret下面放payload，ret前面全部填充，ret到下面的地址。由于限制字符输入为0x40，所以剩下0x40-0x20=0x20的大小可以用。从https://www.exploit-db.com/shellcodes里挑一个小于等于这个大小的shellcode就行，这里用的是https://www.exploit-db.com/shellcodes/46907。结果还是返回一个shell，即执行execve /bin/sh。基本没出现啥意外，很顺利就过了。

* 1. **ret2syscall**

开局遇到个新知识，当32位X86调用系统调用（int 80h）时，会把系统调用号存在eax，剩下的参数按顺序存到ebx，ecx，edx，esi，edi。

X64时，用户态一般函数参数传递顺序是rdi，rsi，rdx，rcx，r8，r9。系统调用内核态是rdi，rsi，rdx，r10，r8，r9。有争议，来源见下方链接。  
（https://blog.csdn.net/weixin\_45574485/article/details/106200229）

此时需要安装一个新工具ropgadget来检索一个程序里可能存在的gadgets（https://www.wangt.cc/2022/02/ropgadget%E5%AE%89%E8%A3%85%E5%8F%8A%E6%8A%A5%E9%94%99%E8%A7%A3%E5%86%B3/）

题目介绍了一个新操作，有时由于开了NX保护，使得不能在内存部分页不可执行。导致不能通过写入shellcode到栈，BSS等区域后跳转执行。这时的办法就是从现有的程序中截取已有的程序片段然后通过反复跳转ret将寄存器的值（系统调用时参数存在寄存器里）改为合适的值，然后int 80h强行日穿。

* 1. **ret2libc**

在进行这个学习之前，先学习了PLT和GOT两个表。这两个表是ELF文件进行动态链接时重要的两个表。其学习总结写在第五章基础知识部分。

需要注意的是，本次函数调用不是系统调用，因此参数实际上是写在栈里的。同时由于我们跳转前汇编代码不可能为我们进行push parameter;push ret的操作，而这是函数栈固定的结构，必须实现。所以我们需要手动完成，但system返回地址随便填就行。（栈结构变化：buf->ebp->ret->parameter变成buf->四字节填充->system函数地址PLT表首地址->随意的返回地址->参数（即/bin/sh地址））

① ret2libc1只要找到system函数地址，找到/bin/sh地址就可以顺利完成。

② ret2libc2中，没有现成的/bin/sh字符串可以用了。但system函数还是有。所以得自己手动输入字符串”/bin/sh”，然而程序里只有一个gets，那个gets必须用来搞溢出，构建栈内环境。所以还需要再执行一遍gets函数，并且另外找个缓冲区写入”/bin/sh”字符串。在bss段注意到一个buf2缓冲区，可以写在那里面。要注意的是，调用gets函数后也要在后面写返回地址和参数（即buf2地址）。而返回地址返回的程序段需要先把gets在栈里的参数（buf2地址）先弹出去，然后再执行ret命令，所以要用ROPgadget找一个pop\_ret的指令地址。把gets的返回地址指定为那里。所以最后栈环境为buf->随便填充4字节->gets的PLT表地址->pop\_ret指令地址->buf2地址->system的PLT表地址->随便地址->buf2的地址。

注意，虽然实际测试pop\_ret可以pop ebp后ret。但这会把ebp改成buf2的地址，这会带来不必要的风险。所以还是pop别的比较好，例如本例中就是ebx。

注意，buf2不能用buf代替，因为在第一次ret前就会先leava，此时esp就指向了ret的地址，buf理论上被释放了。后面还要执行gets函数和system函数，大概率在用栈的时候把buf又覆盖掉。实践后证明确实会寄。

③ ret2libc3甚至去掉了system的函数。但理论上，system函数属于libc，而在libc.so中，所有函数的相对偏移是确定的。所以可以通过got表泄露。即输出某函数的got表项内容后确定用的libc，进而知道system函数地址。当然，由于延迟绑定，需要泄露运行过的函数才能让他在got表中的地址是有用的。

上述操作比较麻烦，可以用工具LibcSearcher代替。

思路是先获取该ELF文件的\_\_libc\_start\_main函数在got表里的值（即其地址），由于这个上来就会执行所以got表里是真实的，顺带获取puts在PLT表里的入口地址和main函数的地址。然后首先把ret用puts函数顶掉，函数返回地址设为main的地址，参数为\_\_libc\_start\_main在got表的里值，即真实地址。然后将这个真实地址用LibSearcher搜索，确定该函数所处的libc。用\_\_libc\_start\_main的真实地址减去在libc里显示的地址就可以获得函数偏移地址。用该地址加上libc里system的地址就是system的真实地址，用此法顺便获取/bin/sh的真实地址。然后因为main又执行了一遍，所以缓冲区再溢出一次，这次ret用system的真实地址替换，返回地址随意，参数为/bin/sh的地址，就成功了。

实验过程中出现发生LibSearcher找不到libc库的情况，参照下面这个链接解决这个问题,主要是要更新libc库。（https://www.jianshu.com/p/a8b3ed1792fa）

实验过程中，最后代码里buf填充不是填充ret-buf个数据了，而是ret-buf-8个数据。理论上是通过GDB跟随调试然后在那个位置用EBP+0x4-ESP-0x1c算出来的，但到底为什么和上面有差距仍然不清楚。

* 1. **train.cs.nctu.edu.tw: ret2libc**

本地实际上是很简单的一个题，他甚至直接给出了/bin/sh和puts函数的地址。然而我卡了半天是因为我想用(5)的万能思路过一遍，但是我在第一次发送payload打印libc\_start\_main的地址时，用的是send而不是sendline，结果scanf死活不结束，我后来才意识到这个问题，最后万能思路试过了。实际上更简单的思路是直接用puts来找libc，顺便算偏移地址。就不用重启main，一次就搞定了。

* 1. **train.cs.nctu.edu.tw: rop**

相当有意思的一道题，并没有直接给源程序，而是远程ns连接后给了该题目所有能用的gadgets，然后让自己拼接顺序。需要注意的一点是，其中几个16进制数实际上可以用p32转换为字符串，通过这个就找到了/bin/sh，这玩意要放到ebx里去。System的32位调用号是11，所以eax里要用一系列操作凑出11，ecx和edx都为0。同理要凑出来。

我是先在纸上写下了凑出四个寄存器必要的步骤，然后给他们排序。由于存在互斥的关系，即有的寄存器操作完成后里面的值不能在动了，但操作别的寄存器时会用到之前写好的寄存器，那基本意味着这两个顺序要掉过来，所以顺序基本上是固定的。同时也有在搞一个寄存器时顺便把另一个寄存器搞好的操作。总之我最后的顺序是9,9,1,10,9,3,3,12,4,8,8,8,8,8,13,13,4,2,0。前面搞的一个9是为了堆栈平衡，别让栈退过头了，第二个9是为了让/bin/sh的字符串和之前的栈空间间隔一下，免得如果之前栈空间也可以转成一个字符串，那就可能转成/bin/sh\001(https://baymrx.me/posts/4727b461.html#%E5%85%B7%E4%BD%93%E5%AE%9E%E7%8E%B0)

* 1. **2013-PlaidCTF-ropasaurusrex**

不难，ret2libc题。日常看看有没有现成的system或execve指令，没有。看看有没有/bin/sh，没有。看看bss或栈能不能执行，不能。看看有没有int 80h，也没有。那就ret2libc了。题目给了个read和write函数。Gdb时不能直接b到main和start有点小搞到了我，不过通过\_\_libc\_start\_main设个断点。而且事后ELF.symbols也搞不到main和start。据说是可以把程序strip掉，我也不是很懂。反正既然如此那就通过IDA直接搞到获取程序里代码的地址然后写死在python里就行了。思路就是先搞到\_\_libc\_start\_main的got地址，构造payload调用write函数把这个打印出来。获悉对应的libc，然后再获取偏移，搞到system和/bin/sh的真实地址，再构造一次跑就行了。

* 1. **Defcon 2015 Qualifier: R0pbaby**

目前为止做的最难的一个ret2libc题。程序不再是以前高度精简的形式，而是一个反编译后依然不小的程序。同时还加上了FORTIFY保护，这是一个会检查是否存在缓冲区溢出的保护。总之会让缓冲区溢出的过程变得困难了。

好在程序本身的功能就是提供各个函数的地址。这为我省下了不少功夫。System函数的真实地址程序可以直接返回给我，然后通过Libcseacher找到system的在libc中的地址，两者相减得到偏移地址。之后把/bin/sh的地址也搞出来。

要注意的是，这是64位程序，所以函数参数在寄存器里，故/bin/sh需要放在edi里。所以还要用GOPgadget找个pop edi；ret的代码。这个代码地址找到后一样加上偏移就得到了真实地址。

函数溢出的地方在最下面的一个memcpy函数那里，从IDA里注意到这个savedregs居然就在rbp旁边。所以只要用8字节覆盖了ebp后就可以写入ret了。



即便如此，最后还是搞不到shell，按照某个博客的操作，加了一个ret就行了，原因是运行system时有个movaps函数，他要求rsp的地址必须是16字节对齐，而实际测试发现正好没对齐16字节，所以没过。而解决办法也很简单，在ret后再加一个ret就好了，这样pop\_ret后就会跳转到一个ret指令，同时因为之前没对齐，就意味着差8字节，多ret一次等于多pop一次，esp就少了8字节，然后就正好对齐了。问题就解决了（https://baymrx.me/posts/cb6f85#%E9%A2%98%E7%9B%AE%E5%88%86%E8%A7%A3）

学到一个gdb找偏移地址的新操作，虽然IDA很方便。但gdb总归来的保险。首先pattern\_create n生成长度为n的字符串，输入到程序，运行。程序八成会中断在ret，则rsp指向ret的地址。此时x /gx $rsp 用16进制8字节显示rsp里面的值。复制出来，用patter\_offset string把这个值输入进去，就会显示出溢出地点到字符串最初位置的偏差值，也即缓冲区首地址到ret首地址的偏差值。

后续补充：gdb找偏移地址的操作在64位和32位可能有不同，64位上面的就很顺利，在后面一个32位程序（pwn200-no relro）里使用时，发现esp已经指向了地址的下4个字节，eip里已经被覆盖了。但64位时就不是。

* 1. **总结与反思**

基础ROP做完了，简单来说学了ret2text，ret2shellcode，ret2syscall，ret2libc。目前来看栈溢出大体思路如下。  
 ① 首先checksec看看程序的限制和位数。如果没有nx限制则八成可以栈执行或bss执行。32位程序和64位用IDA不一样，并且函数执行时参数存放位置也有变化。

② 用IDA看看程序有没有现成的system或execve能用，这样就可以直接ret2text。

③ gdb看看栈和bss之类的地方有没有执行权限。有的话如果空间够大，可以直接ret2shellcode

④ 用ROPgadget看看有没有int 80h。有的话可以考虑拼凑gadget操作一下ret2syscall

⑤ 都不行就直接ret2libc，一般\_\_libc\_start\_main都会执行，所以如果其他函数都用不了就用这个来找libc，之后走流程找system和/bin/sh就行。

* + - 1. **中级ROP**

1. **Ret2csu**

介绍了用如何利用\_\_libc\_csu\_init构造gadget。本题中基本思路是利用上述函数中的几个gadget来操作寄存器，进而执行想要执行的函数。之前学的ret2libc一直操作的都是32位程序，这时参数存在栈里，所以好操作。但64位程序时参数就会依次存在rdi，rsi，rdx，rcx，R8，R9。这意味着我们需要通过pop操作来依次给上述寄存器赋予参数。但如果源程序里没有合适的pop操作，那就寄了。而ret2csu中利用的\_\_libc\_csu\_init就是解决这个问题的。同时该函数的gadget中还非常好的已经写好了一个call调用，因此只要合适的操作寄存器，就可以想执行什么就执行什么。

不过要注意的是，这个函数的gadget最多操作到edx，所以需要执行的函数最多3个参数。

中途遇到的问题包括

1. 想要用system来搞到shell，但是失败了。根据ctf-wiki上解释，是环境变量问题。所以改为execve
2. 想要调用execve时直接把参数写为libc中/bin/sh的地址，但是失败了，因为使用ret2csu操作时，给edi赋值最多赋低32位，高位固定是0，这是汇编代码决定的，没法修改。因此只能把/bin/sh写入到高32位全0的区域来执行，最后选定了bss段。注意调用read函数写时，字符串还要加\0作为终止符。
3. 当利用\_\_libc\_csu\_init中的call执行函数时，注意到r12中的值应当是存有执行函数地址的地址，因为代码是call qword ptr [r12+rbx\*8]，故r12值应当是某函数got表地址或存有函数地址的地址
4. **Pwn-100**

值得纪念的一题，第一次写出和看过的答案不一样的解法。本体唯一要注意的是puts函数在输出到\x00时就停了，而且puts会在结尾自带一个\n。由于puts可能会提前停，导致得到的地址不一定是8位的。所以需要自己手动补0。不过也有运气成分在里面，因为万一停了之后的地址有非0的就寄了。不过还好没有发生。要注意的还有在函数没调用过plt进got前，got里面的地址是假的。plt有个查询函数得先过个流程。

还有一点就是别信flat，这次flat就翻车了，还得是手动p64添加来的好。

上文利用的\_\_libc\_csu\_init这次没用到，因为puts函数和system函数都只用一个参数，而且正好有一个pop edi;ret可以用，就不需要再init里面拿了，顺带一提，这个函数在这个程序中名字就叫init而不是\_\_libc\_csu\_init，一开始我还没找到。

1. **2016 华山杯 SU\_PWN**

实在找不到源文件，贴一个writeup在这里(https://atum.li/2016/09/24/supwn/)

1. **ret2reg**

没给例题，但看描述不难。看起来就是64位版的ret2shellcode。首先看溢出函数返回时哪个寄存器指向了溢出缓冲区空间。然后用ROPgadget找call reg或jmp reg。最后往reg指向的空间写入shellcode即可。但要确保有执行权限，但一般是在栈上。

1. **Brop**

目前为止最复杂的一个，因为其基本前提是无法直接访问到源程序，所以需要用remote进行访问。这个攻击手法需要有两个基本前提

1. 源程序必须有栈溢出漏洞，这个前提是显然的。
2. 服务器端进程在崩溃后会重新启动，且重启进程的地址和原来的一样。（即使有ASLR保护，但也得一样）。目前nginx，mysql，apache，openssh可以满足。

其基本思路如下

1. 通过暴力枚举来得到栈溢出长度
2. 如果有canary防护，则通过逐步解密的方式泄露canaries信息
3. 通过blind rop（BROP）来控制函数参数，并通过puts函数等输出函数
4. 利用输出函数dump出整个程序，便于找到更多gadgets，从而写出最后的exploit。

上面是这么说的，我来复盘一下我的实操。

通过checksec发现给出的样例程序没有canary保护，所以第二步可以去掉。首先还是通过暴力枚举得到栈溢出长度，方法就是从1字节开始往里写，写到程序崩溃（该样例中的表现为没输出字符串“No password, no game”）为止，意味着ret被覆盖了，所以此时字符串长度-1就是需要覆盖buf首地址到ret首地址的长度distance。

然后需要寻找stop gadget，stop gadget就是可以让程序无限循环不崩溃的一个gadget。可以通过distance\*’a’+addr+’0x0’\*10或distance\*’a’+addr来得到不让程序崩溃的addr，即stop gadget。实操的时候用的是后者，但后来测试前者也可以，只是前者得到的地址会是比较后面的，可能是会严格一点。

找到了stop gadget，就等于有了终止标志，可以开始寻找合适的工具了。这里采取的工具就是\_\_libc\_init\_start的最后那6个连续的pop。这里用IDA作弊看了一眼，确实是pop而不是之前遇到过的mov，比较幸运。为了找到这6个连续的pop，只需要通过distance\*’a’+addr+b’0x0’\*6+stop\_gadget+b’0x0’\*10就可以找到（实际操作中，出现有的地址输入进来后脚本卡在recv的情况，理论上要么是无限循环了，要么就是在recv处莫名阻塞了，所以要加timeout，且当recv超时后返回值是空字符串，故可以用len判断返回值长度来pass掉这个地址）。然而这样并不保险，因为addr有可能是个stop gadget,所以还需要check一下。可以通过构造distance\*’a’+addr+b’0x0’\*18来检测，如果这样都不崩，那说明addr是stop gadget,反之就意味着找到brop gadget了。

找到了brop gadget就意味着找到了pop rbx, pop rbp, pop rsi, pop rdi。最重要的rsi和rdi是通过brop gadget+7和+9凑出来的。rdi是第一个参数，rsi是第二个。Rdx没能直接搞到，但理论上可以通过strcmp函数来控制，不过本题中没用到。

有了工具，就要准备dump出程序了，这需要puts函数的支持。为了使用puts函数，需要先找到他的plt。这可以通过执行puts函数的功能来实现，由于已知0x400000以字符串形式输出开头一定是\x7fELF，所以可以通过以下payload来找出puts。

payload=distance\*b’a’ + pop\_rdi\_ret + 0x400000 + addr + stop\_gadget

如果addr正好找到了puts\_plt，那输出的startwith一定是\x7ELF。基于此puts\_plt就找到了。（不过我实践中不是正好是puts\_plt的起始地址，而是这个地址前面几个字节，但影响没那么大，只是后期找got时会有干扰）

找到了puts\_plt就可以开始dump出程序了，一半打印到0x401000就够了，可以通过distance\*b’a’ + pop\_rdi\_ret + addr + puts\_plt +stop\_gadget循环输出dump出所有结果。要注意的是，addr每次更新应当是前次的输出字符串的长度，即前次打印的字节数。

在实际操作中，puts会自动加上\n，因此可以通过content = content[:content.index(b"\n")]来得到正确的输出。注意由于输出是字节流，因此一定要加b。并且当puts遇到\x00时，就直接不输出了，所以如果遇到空字符串，那就要手动让content = b“\x00”，不然有的字节不输出最后地址会错位。同时因为未知原因（可能是后面的地址不能读取），当打印第0x401000处的字符串时，脚本就卡在了0x401000处反复读取，因为我设置了如果报错时content=None,此时直接continue，addr不增加。所以出了问题，故需要设置一个count计数器，如果addr连续三次不变化就break。

把打印结果在一个字符串里拼接起来，并以二进制形式输出到一个文件里。再用IDA以二进制形式打开文件，注意要通过edit->segments->rebase program 将程序的基地址改为 0x400000。然后跳转到之前put\_plt的地址，把下面几个字节按c反汇编，就能找到got地址。至此puts的got地址也找到了。

之后只要按正常流程，用distance\*b’a’ + pop\_rdi\_ret + puts\_got + puts\_plt + stop\_addr 来dump出puts的真实地址，用libcsearch查出libc，计算偏移，顺便搞到system和/bin/sh的地址，再次构造distance\*b’a’ + pop\_rdi\_ret + binsh + system + stop\_addr就可以搞到shell。

但实际操作中，有个两个问题，一是如果stop\_addr就是main的入口地址，那实际上程序就不用重启，可以直接继续执行。反之就要重启程序。二是当我实际操作是，上述payload并不能拿到shell，需要改为distance\*b’a’ + pop\_rdi\_ret + binsh + ret + system + stop\_addr才行。这个问题之前遇到过，是因为system要求rsp对齐到16字节，所以没对齐肯定是rsp对齐到8字节了，只要再ret一次就可以了。而retn的地址可以直接通过brop\_addr + 10得到，比较方便

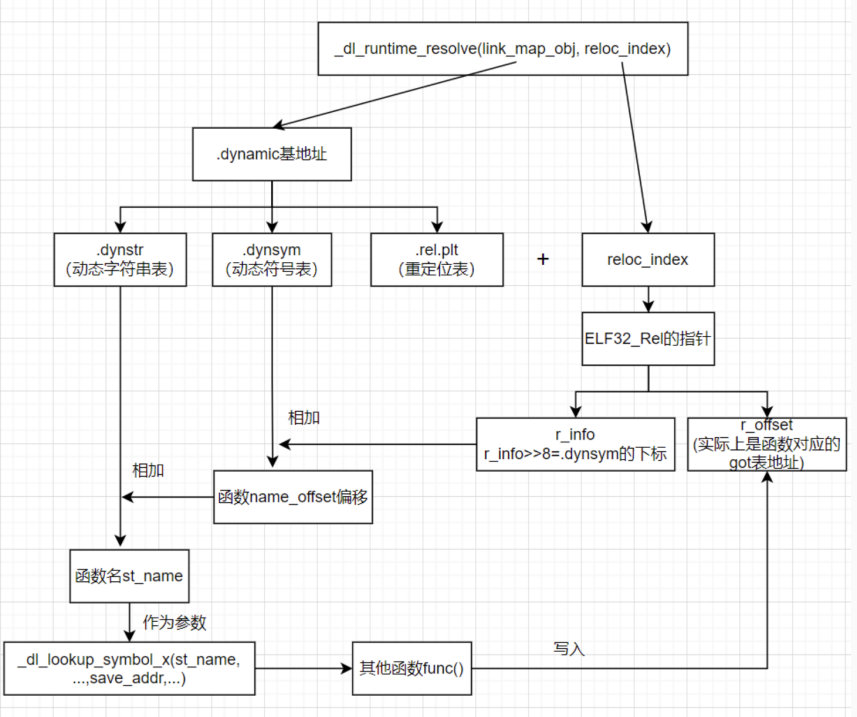
* + - 1. **高级ROP**

1. **main\_norelro\_32**

原理叙述更加详细的解释见两个博客，wiki给的有点抽象，不过也能用（https://blog.csdn.net/jzc020121/article/details/116312592）

（https://pythontechworld.com/article/detail/iAD6PiSHmozZ）

这题做之前必须先理解ELF文件的大致结构和动态链接的全过程，为此学习了很多天，直到今天才开始正式做题。相关基础知识见第五章，或是直接看上面的博客链接。这里截取一个比较重要的图片



该题关闭了relro机制，这意味着.dynamic是可写的。根据前置知识，我们知道,\_dl\_runtime\_resolve进行解析时，最终确定解析函数的方法还是通过获取.dynstr中的目标函数的函数名，然后进行比对。所以攻击方法油然而生。

1. 首先修改.dynamic节中的.dynstr节的地址，将其变成一个我们伪造的地址。
2. 复制.dynstr节的内容，将里面某个函数名（例如read）改成system，然后把.dynstr的内容通过read等方式写入到伪造的地址处。
3. 挑一个风水宝地（可写）的地方写入字符串’/bin/sh’
4. 跳转到被替换的函数（例如上述的read）的plt表的第二条指令处（push），然后走流程调用\_dl\_runtime\_resolve进行函数解析，从而执行system函数。

思路不难，总结一下做题时学到的东西与发生的问题。

首先学会了一个新操作，可以一次性构造好ROP chain，然后哗啦啦一块读进去（不过得保证源程序能读这么多字节），就是pwntools的ROP函数，该函数可以从程序中自动找到想要构造的函数，并把写入的东西都准备好。总之可以看下面这个博客或者文档（https://www.csdn.net/tags/MtTaEg2sMjkwNjIzLWJsb2cO0O0O.html）

本次遇到的问题有几个，一是我脑子抽了，忘了栈里参数前还得写个函数返回地址，我还在奇怪了很久为啥要插入一个rop.raw(0xdeadbeef)。

二是在复制.dynstr节内容，并写入到伪造地址时，第一次没调好地址，导致.dynstr节太长了覆盖了got表的部分内容，read正好被盖住了，结果出了一些毛病。后来看了一下got后面没重要内容，就在got后面写入搞定了。

1. **main\_partial\_relro\_32**

本题禁止修改.dynamic，因此总体思路是伪造.rel.plt, .dynsym, dynstr等表使得运行函数。

由于函数栈给的空间太小，因此首先教了一个栈迁移的操作。下面迁移的目的地址均为bss段+0x800，这里简称targ\_addr，这个0x800是给栈预留的空间。

栈迁移的原理如下

1. 找个pop\_ebp\_ret的函数或在用垃圾数覆盖栈时少覆盖4字节，ebp的位置用targ\_addr-4覆盖。这样就可以在leave或pop ebp时首先把ebp转移到targ\_addr-4。为啥要-4后面会解释
2. 在ret的位置用read函数覆盖，这样就可以在新栈位置targ\_addr写入数据，当栈转移后就可以无缝衔接继续执行我们的程序。
3. Read的返回地址用连着三个pop后ret或者别的方法跳过read的参数，因为得执行后面的东西。
4. 找个leave\_ret的代码片段，由于leave等价于mov esp,ebp;pop ebp。所以这个操作后，esp就被成功转移到了ebp的位置，即targ\_addr-4，但由于还要pop一个ebp，所以这个操作执行完成后esp就真正指向了targ\_addr了，栈转移就完成了，当然ebp就飞不见了，不过他不用管。要管也可以，在第②步时将第二个参数设为targ\_addr-4就可以，这样就在执行write时可以顺带的把targ\_addr-4写好，即写好ebp的新值。

整个题目都要栈迁移，上面的是通用过程，实践中用的是rop.migrate()实现的，方便很多，但他要求程序里有可以利用的代码片段，没有就寄了。

后面就按顺序依次介绍不同的stage，从开始到后面会用手工的方法逐渐深入底层，可以更深刻理解ret2dlresolve

1. stage 1：

通过gdb pattern\_create和pattern\_offset可以轻易知道栈溢出大小为112字节。这一关我们想怎么操作怎么操作，目的是写下一个/bin/sh后用write打印出来。但栈转移还是要的。因此还是首先栈转移过去。这时我们可以直接用write的plt表项来执行write函数，后面就得手动日了。

为了避免误会，这先声明几个默认条件，首先是转移前的read函数设定是会读取100个字节，这是后面数字100的来源。其次就是默认/bin/sh字符串会写在base\_stage+80的位置。最后就是栈大小设定是0x800，不过这个对后面影响不大。

所以转移后的payload如下：

rop.write(1,base\_stage+80,len(binsh)) # 把/bin/sh放在80字节的位置

rop.raw(b’a’\*(80 - len(rop.chain())) # write和binsh之间用a填充

rop.raw(binsh)

rop.raw(b’a’\*(100-len(rop.chain())) # 栈迁移时read了100个字符，这里填满

1. stage 2：

上一关就练习了一个栈迁移，现在得开始手操各个动态表了。我们知道，延迟解析调用解析函数时，有两个参数，一个时linkmap，一个是reloc\_offset。前者先不管，后者在32位下有一个等式：reloc\_offset + .rel.plt = ELF\_rel。

即.rel.plt首地址+这个偏移就是目标函数的ELF\_rel结构。（64位下reloc\_offset不是偏移而是下标）。这个结构体里有两个成员，一个是r\_offset和r\_info。前者的值等于目标函数got表地址，后者在32位下的高3字节为该函数在.dynsym里的下标。这一关我们先只关注r\_offset，即我们要手动解析write函数然后运行一遍。

为了实现上面这个目的，我们得先搞到plt0的地址，这个地址值正好是push link\_map的那条指令。所以我们要手动push r\_offset进去。下面这条指令拿到了plt0的地址。之后不再重复。

plt0=elf.get\_section\_by\_name(‘.plt’).header.sh\_addr

其中sh\_addr是section header table表里每个结构体的成员，标识节的起始地址。

搞到write的reloc\_offset的方法是首先搞到.rel.plt表的内容，然后从用write.got的值从里面匹配，匹配成功的那个就是write的r\_offset，他的地址减去.rel.plt的首地址就是reloc\_offset，这可以通过以下几条指令完成

write\_got = elf.got[“write”]

relplt\_data = elf.get\_section\_by\_name(“.rel.plt”).data()

write\_reloc\_offset = relplt\_data.find(p32(write\_got))

因此转移后的payload就是

rop.raw(plt0)

rop.raw(write\_reloc\_offset)

rop.raw(0xdeadbeaf) # ret addr

rop.raw(1)

rop.raw(base\_stage+80)

rop.raw(len(binsh))

rop.raw(b’a’\*(80 - len(rop.chain()))

rop.raw(binsh)

rop.raw(b’a’\*(100-len(rop.chain()))

1. stage 3：

上一关还畏畏缩缩的只能拿现成的.rel.plt。这一关就干脆直接自己造一个假的rel.plt放在新栈里，这样就摆脱了.rel.plt的卡脖子行为。

为此，我们首先需要完成几个目标。

由于reloc\_offset+.rel.plt就等于目标函数的ELF\_rel结构体。虽然我们自己造了一个.rel.plt，但毕竟.dynamic里没变，计算机是不认得，所以我们只能退而求其次的改变reloc\_offset。这样就简单了，我们可以在栈里写好ELF\_rel后用他的地址减去.rel.plt就得到了新的reloc\_offset，当然不避免的是reloc\_offset会很大。

我们事先设定新的.rel.plt就从base\_stage+24开始，因为前面的还是得写write函数的参数等杂七杂八的。而且由于我们只用利用一个write，所以我们不需要把整个.rel.plt都伪造过来，只要搞一个write的就行。我们可以通过以下几行代码得到reloc\_offset

relplt\_addr = elf.get\_section\_by\_name(‘.rel.plt’).header.sh\_addr

fake\_relplt = base\_stage + 24

write\_reloc\_offset = fake\_relplt - relplt\_addr

现在我们地址已经造好了，只要往里写东西就行了，ELF\_rel结构体里有两个成员r\_offset和r\_info，前者等于write的got表地址，获取方法stage 2里有。后者只能通过IDA或gdb亲自去看看了，最后得到其值为0x607。最低8位的7是固定的，程序运行时会有专门的代码检测这个7在不在。

综上，我们这次的payload也就出炉了。

rop.raw(plt0)

rop.raw(write\_reloc\_offset)

rop.raw(0xdeadbeaf)

rop.raw(1)

rop.raw(base\_stage+80)

rop.raw(len(binsh))

rop.raw(write\_got)

rop.raw(0x607)

rop.raw(b’a’\*(80 - len(rop.chain()))

rop.raw(binsh)

rop.raw(b’a’\*(100-len(rop.chain()))

1. stage 4

理论很简单，实践起来却出问题的一关。

stage 3说道，我们通过伪造.rel.plt解决了他的卡脖子问题，那我们同理是不是也可以伪造.dynsym？答案是肯定的，这一关就是搞这个。

我们知道ELF\_rel结构体里有个r\_info，里面前3字节就是目标函数在.dynsym里的下标。但既然我们要把.dynsym也写在栈里，这也就意味着r\_info也要改，同时由于这个值是下标而不是偏移，所以我们还要确定新.dynsym后把他的地址和真实的.dynsym的地址相减并除以结构体ELF\_sym的大小0x10。

这里有个要注意的点，由于程序是通过下标寻找目标函数的ELF\_sym，因此如果我们的新地址和真实地址之差不是0x10的整数就会出问题，因此我们还需要填充字节来进行对齐。现在我们规定新的.dynsym的地址在base\_stage+32的为止，就贴着新.rel.plt。那么r\_info的求法如下。

dynsym\_addr = elf.get\_section\_by\_name(‘.dynsym’).header.sh\_addr

align = 0x10 - ((base\_stage + 32 - dynsym\_addr) & 0xf) # 取余也行

fake\_dynsym\_addr = align + base\_stage + 32

dynsym\_index = (fake\_dynsym\_addr - dynsym\_addr) // 0x10

r\_info = dynsym\_index<<8 + 0x7 # 0x7是固定的

然后新的ELF\_sym从IDA里看看原来的，复制过来就行。

fake\_dynsym\_data = flat([0x4c,0,0,0x12])

因此可以得到payload

rop.raw(plt0)

rop.raw(write\_reloc\_offset)

rop.raw(0xdeadbeaf)

rop.raw(1)

rop.raw(base\_stage+80)

rop.raw(len(binsh))

rop.raw(write\_got)

rop.raw(r\_info)

rop.raw(align\*b’a’) # 别忘了填充

rop.raw(fake\_dynsym\_data)

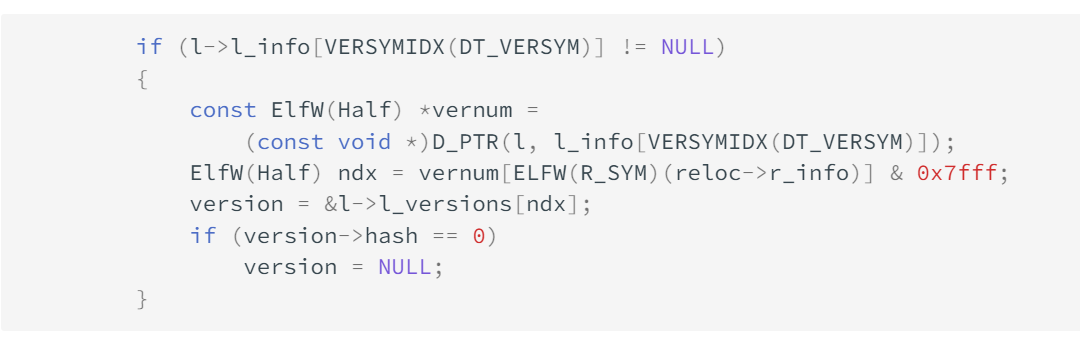
rop.raw(b’a’\*(80 - len(rop.chain()))

rop.raw(binsh)

rop.raw(b’a’\*(100-len(rop.chain()))

按理来说应该这样就OK了，但实践出了问题，ctf-wiki里叙述了问题是什么以及怎么解决的，里面直接贴上了glibc的源码，但很难看的明白，毕竟里面的变量声明和宏都比较抽象，但花了一天时间大致搞懂了修改逻辑。

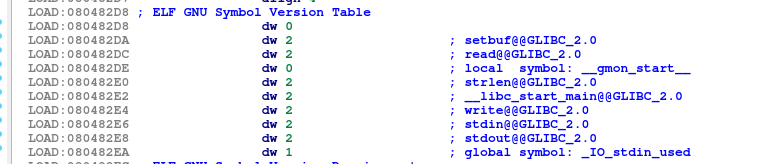
首先先把源码贴上来，不需要全部弄明白，只要知道大致含义即可。



出问题的是这里的ndx，ndx的地址实际上等于.gnu.version的首地址加上fake\_dynsym的偏移下标\*2，即下述代码。



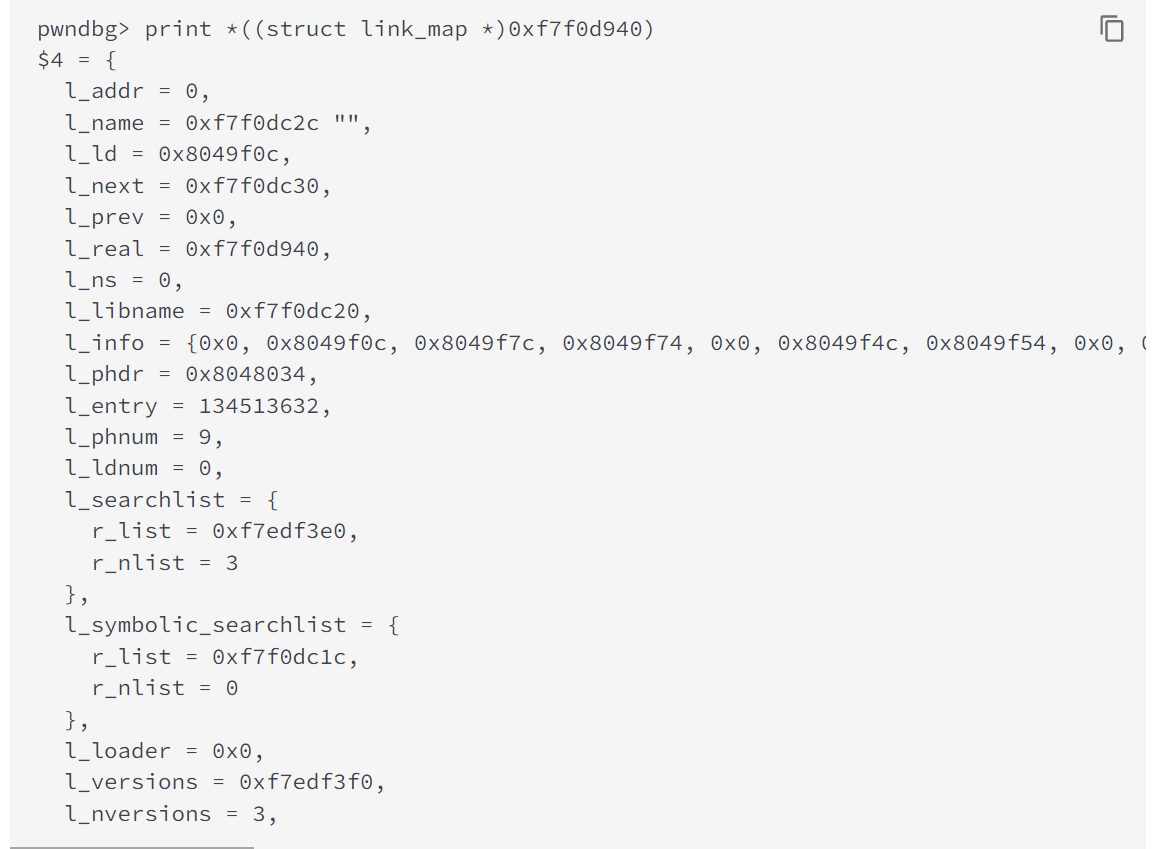
这里vernum实际上就是.gnu.version首地址，里面每个元素都是2字节的，如下图中显示的，里面显示了一些版本号，这些版本号会在后面起到作用。ndx赋值时表达式右边的的ELFW(R\_SYM)(reloc->r\_info)理应等于index\_dynsym，即dynsym项偏移地址除以0x10，总之就可以得到ndx\_addr。



后面我没亲自去操作一遍（做stage5的时候操作了一遍，和ctf-wiki里数据是一致的），但可以直接跟着ctf-wiki里面解释。ctf-wiki里发现ndx的地址在0x80487a8，通过gdb跟踪发现掉到了.eh\_frame节里，这个不重要，重要的是ndx的值是0x442c。这个值相当的大。源代码里有一行代码如下

version = &l->l\_versions[ndx]

这行代码是什么意思呢？先解释l->l\_version是什么。l的结构根据ctf-wiki里显示如下图。



很多元素，但都不用管，可以看到l\_versions的地址是0xf7edf3f0，根据代码可以看的出来这是个数组，里头的结构如下。



可以看到里面有三个成员，每个成员而我们的ndx本来应该是0到2，现在却大的离谱了，所以寄了。

但观察源码里，发现当version->hash=0时，version=NULL。根据资料显示，等于NULL是没关系的。（https://forum.90sec.com/t/topic/260），所以我们只要让ndx变成0或1就行。为此我们只要改变程序的base\_stage使得ndx的新值等于0或1就行。0比较常见和方便，所以改成0。

ctf-wiki里找到了一个地址是0x080487C2，这个地址在080487A8（原来的ndx地址）下面而且地址里的值是0（注意ndx是要与0x7fff的，这意味着这是个两字节数，且只取15位）。为了让ndx迁移到这里，我们需要这样调整。

根据公式ndx\_addr=gnu\_version\_addr + index\_dynsym \* 2

则index\_dynsym(变化)=(0x080487C2-0x080487A8)/2

又因为index\_dynsym = (fake\_dynsym\_addr - dynsym\_addr) / 0x10

且index\_dynsym = base\_stage+32

故base\_stage(变化) = ((0x080487C2-080487A8)\*0x10)/2

所以把base\_stage加上上述数值即可。

1. stage 5

stage 4过了stage5就比较简单了，只需要把’write\x00’的字符串写到fake\_dynsym后面，然后修改fake\_dynsym中st\_name的值为该字符串到.dynstr的偏移就行。和stage4基本没啥区别。这次搞得时候顺便把stage4中ctf-wiki的操作复现了一遍，基本上是完全搞明白stage 4了。就是rop.chain偶尔出毛病，又莫名奇妙好了，不知道怎么回事。而且还遇到个a<<8+1时没打括号的毛病，由于+优先级大于<<，所以这等价于a<<9。就寄了。

1. stage6

stage 6也没难度，就是把write的字符串改成system，然后由于write有3个参数，但system只有一个，所以后面所有fake数据地址都要减8字节。但要是在原来的参数位用8字节的a填充也行，后面地址就不用变了。反正后面也就是个数据位，所以填不填充都无所谓。

只是要注意把dynsym的对齐填充补上，这次忘了补导致又懵逼一会，debug了半天才发现是填充问题。

1. 基于工具的伪造——pwntools

ROPUTIL没用过，懒得深入看，直接看pwntools。这里用了一个新函数Ret2dlresolvePayload构造payload。这个三个参数，第一个是elf对象，第二个要执行的函数（例如system），第三个是函数参数。

这个操作中没有进行栈迁移，而是pwntools自动找了一个能用的空间写数据。基本没什么需要操心的按照代码的操作走流程就行。唯一要提一下的是他的payload用了一个flat，里面是个字典。这个字典的键代表着在第几个字节处开始写入值代表的数据。所以raw\_rop会在112个字节处开始写，前面随便填充。而dlresolve.payload会在256字节处开始写。要注意的是，整个程序会有两次read，第一次是程序自带的，第二次是我们字节写进栈里的。但他一次sendline就完事了，因为在源程序里就只能read 256个字节，所以即使只搞了一次sendline，但第一次read就在第256字节就停了。后面还在缓冲区里等着读，而第二次就读到的目标地址里了。这也是键为256的原因。

1. **main\_no\_relro\_64**

很折磨我的一关，这一关事实上思路不是很难理解，就是实践起来出了不少问题，而且哪怕直接照搬ctf-wiki的代码都过不去，最后还是修改一个小细节过去了，虽然我也不知道那一点点字节的区别为什么会造成这么大的印象。

不过正常思路还是要总结的。这一关取消了relro保护，所以可以直接改.dynamic的.symstr地址为我们伪造的地址，然后把read字符串改成system就完了。总体思路就是这么简单，但有几个实践过程中要注意的点。

1. 这次bss的起始地址为600B40，所以不能大张旗鼓的直接+0x800给栈赋个很大的空间，而且事实上，之后的代码和数据都得放在这个栈里，所以实际上很挤。而且哪怕还剩下0x4C0的空间，因为还要往下写0x1a0个字节，所以实际上只能在0x320处设为base\_stage。而且bss头部，即这个往上还得放fake\_symstr表。不过因为不知道什么原因，可能是ctf-wiki精心构造的结果，只要不是+0x200就崩了。
2. 64位时，调用system参数得放到rdi里，而且还不能用csu里的rdi，因为那个只能放低32位。所以得自己抓一个pop rdi的gadget。
3. 本来可以利用源程序的read同时实现给迁移栈赋值并迁移两个目标，但由于只能源程序的read只能读取256个字节，实操之后发现想实现两个目标正好差几个字节，因此上述目标只能分两步走。

综上，实现流程大致是，先用自己的read给迁移栈安排栈里的数据，然后再ret回vul函数多利用一次他的read实现栈迁移。之后在迁移栈里完成修改.dynamic,写fake\_dynstr和/bin/sh的几个目标，并且在最后通过调用read的plt[1]（即PUSH）来调用system。

虽然说的很好听，但实际操作起来出了很多问题，ctf-wiki里提到了16字节对齐问题，我也发现了，但ctf-wiki里的方法根本不行，虽然理论上看着可行，但用了就炸，最后没办法我用ret对齐，莫名奇妙就过了。时至今日依然不知道为什么。这题花了我很多时间，但未解之谜是我见过最多的。只能说奇怪。

1. **main\_partial\_relro\_64**

这几题没动手，有各种各样的原因，但主要还是懒癌犯了。而且也想赶一下进度。看代码还是看得懂的，思路也看的明白，但自己写起来就容易遇到很多问题导致一天就过去了。主要是上一题太折磨了，给我整怕了。如果按照那个效率那很难完成预期的计划，而且这里ctf-wiki里也遇到了不少问题，所以就以看为主。但还是可以总结一下思路和实现。

根据ctf-wiki里，x64里有以下几个变化

1. .rel.plt改为了.rela.plt，两个结构体的区别是多了一个成员。并且后者每个成员都是8字节。同时指向.rel.plt的reloc，从x86的偏移改为了在.rel.plt中的下标，每个结构体24字节。r\_info则为高位4字节指向在.dynsym中的下标。
2. .dynsym中的顺序调节了一下，但影响不大。

他这次攻击主要分为了两个方法，一个是leak（link map）的方法，一个是no leak。先说leak。

1. leak的步骤比较直观，这次甚至没有栈转移。首先是日常定义工具，即万能执行函数csu，然后定义了一个函数ret2dlresolve\_x64，功能是输入fake数据存储地址，执行函数名（system），被顶替的函数got地址（write）。之后层层伪造.dynstr, .dynsym, .rela.plt数据。把他们打包成resolve\_data返回。同时把plt0和reloc\_index打包成resolve\_call返回。
2. 之后栈上正常操作，先调用read把resolve\_data写到fake数据存储地址，然后调回vul函数重启
3. 再次调用read把/bin/sh写到风水宝地后，调用vul重启
4. 最后栈里存个pop\_rdi; ret的地址，把binsh的地址pop到rdi中，最后返回到resolve\_call就行了。

过程很流畅，理想很丰满，但现实很骨感。按理来说上述流程应该没什么问题，但是在ctf-wiki中遇到了如下问题。

1. 获取版本号出错
2. 环境变量被覆盖

前者经过论证，不可能通过调节栈结构来让映射地址位于内存映射中，于是决定泄露link map地址， 将l->l\_info[VERSYMIDX(DT\_VERSYM)] 设置为 NULL来跳过获取版本号的代码。其中VERSYMIDX(DT\_VERSYM)的值是通过汇编里观察得到的。

后者原因是栈覆盖的太长了，这个在以前遇到过博客讲过，可以见7月3号的进度总结。解决思路是尽量不破坏环境变量或把他们变成NULL。所以首先是收缩ROP长度。wiki采用的方法是先把写resolve\_data和/bin/sh的操作合并，然后又试了一次，发现被污染的数据只有0x61，即填充用的’a’。由于不知道是哪里的填充，于是把所有填充都换成’\x00’，即改成NULL解决了。

no leak的办法主要是直接伪造一个link\_map，其主要思路是先让\_\_builtin\_expect(ELFW(ST\_VISIBILITY)(sym->st\_other), 0)不等于0，这可以通过设置 sym->st\_other 不为 0来实现。然后修改符号绝对地址value，其计算公式是value = l->l\_addr + sym->st\_value。而其中l->l\_addr是link\_map首元素，

这里解释以下l\_addr和st\_value。根据我查阅的资料（ctf-wiki里的execute->ELF）和推测，l\_addr的原意应该是函数的内存偏移-文件偏移。而st\_value则为函数的文件偏移。因此相加可以得到函数的内存绝对地址。

而此处的利用方法是将l\_addr设置为目标函数（system）的内存偏移-待替代函数(\_\_libc\_start\_main)的内存偏移。st\_value则设置为\_\_libc\_start\_main的got表内容，即其的内存偏移。这样就得到了system的绝对地址。这里不一定非要是\_\_libc\_start\_main,也可以其他已经解析过的函数，只要got表里的值是真实地址就行。

然后就是伪造link\_map的事情，这里我大概看了一下，主要是有一些细节要处理，虽然没怎么看明白，但就当我大概悟了，这里就过了。

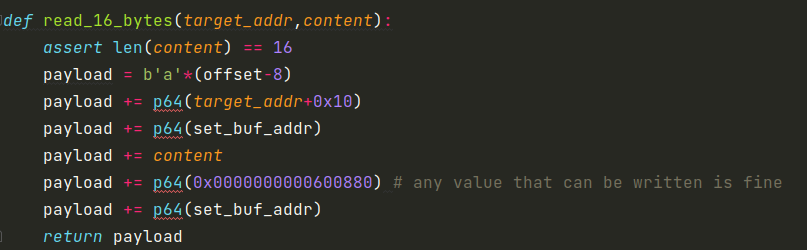
1. **2015-hitcon-readable**

我是傻逼艹，这个题目不难，但硬是卡了我5个小时，只因为一个弱智问题。

但流程还是要正常总结的，过会再说我有多傻逼。

这题目是64位的程序，没有no relro保护，而且bss下面空间很大。所以思路很简单，只要直接伪造.dynstr后修改.dynamic就好了。唯一的问题是本题给的溢出空间很小，源程序只能read 0x20个字节，其中0x10个字节要用来填充缓冲区，所以只有16个字节分别用来填充rbp和ret地址。

但好在观察汇编，发现源程序是用rbp-0x10得到缓冲区首地址的，所以只要控制了rbp，还是想往哪写往那写。写完一次后让ret回到修改缓冲区首地址的那行汇编代码就可以无限循环写数据。唯一的问题只有每次都得写满32字节。根本原因由于leava的存在，rbp会把整个栈给拽过去，同时让rbp的低8字节为ret，高为buf。所以循环节是这样书写的。

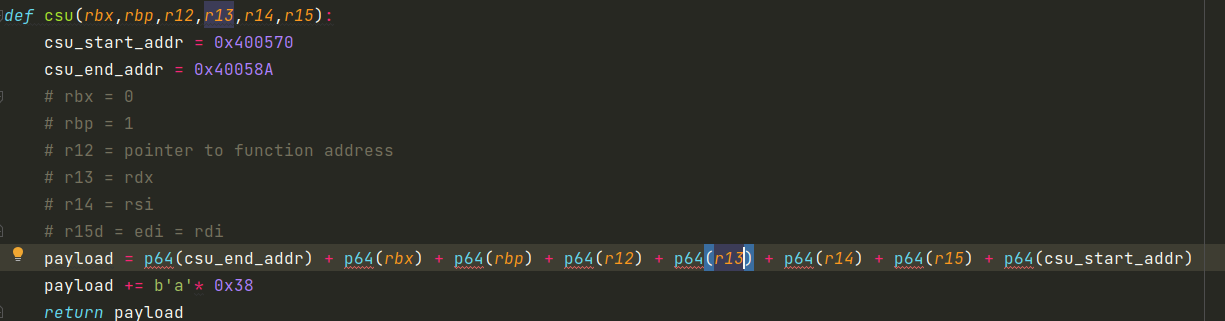


第一次填充时，rbp用target\_addr+0x10顶掉，+0x10是汇编里因为rbp-0x10才是缓冲区首地址。ret用设置buf首地址的那行代码（lea rax, [rbp+buf]）地址顶掉。这时ret后，rbp=target\_addr+0x10，rsp不重要。此时再次写入数据，则target开始16字节为content，rbp被新的，随便一个能写的地址覆盖，rbp下8字节被set\_buf\_addr覆盖。当执行leave时，rsp被拽过来，rbp随便飞了，而ret就又ret回去了，这就完成了一次循环。自始至终rsp都不重要，因为用不到，而rbp在第一次设置好后就实现了他的价值，可以乱飞了，当然目标地址要能写，不然就没下次了。

由于这个操作必须写满32字节（要ret的嘛），所以不能用来改.dynamic。改那个还得自己在新栈里整个read，这是我噩梦的开始。

ctf-wiki的思路是，直接在新栈里把一堆read写好，包括改.dynamic, 伪造.dynstr和写入/bin/sh。而我则是考虑到后两者可以用read\_16\_bytes来写，所以就后面两个大表就用read\_16\_bytes循环搞定了。

然后就是用csu写个read修改.dynamic，改完了就pop\_rdi，放个binsh首地址，ret对齐一下，放上read.plt[1]就完事了。事实上这些都没什么问题，问题出在csu函数。



这是我的csu函数，实际上函数没什么问题，问题在于我的使用上。

可以看到，r13对应的是rdx，也就是第三个参数，而r15对应的才是第一个参数。但我之前顺着写写习惯了，就正好写反了。导致我debug了五个小时，怎么都想不明白为什么read函数刚进去就退出去了，做了各种实验，甚至怀疑过前面输入流里还有数据没读出去，但就是没想到我参数写反了，是之前的惯性思维导致的，因为上一个做的题目还是r13对应rdi，r15对应rdx，写起来就很优美，完全没想过这个顺序还能换的。

总之很傻逼。

1. **2015-hitcon-readable（方法2）**

该方法未经过实践，但理论可行，此处记录在ctf-wiki里对代码的解读。

理论上，libc中肯定会实现syscall函数，因此只要找到该函数的地址，并覆盖在read的got表上，就可以执行系统调用execve，设置好/bin/sh的参数后就完成了。

为此首先要爆破得到syscall的地址。由于read函数和syscall都在libc里，所以只有最后8位不同。因此可以修改read的got的低8位，遍历寻找syscall。由于遍历利用的还是read函数，所以其返回值存在rax里，返回值为写入的字节数。又因为write的系统调用号为1。因此假设syscall找到了，这时由于read修改的是8位，因此read返回1到rax，如果再次执行read的got的地址，那就是syscall 1.即调用write函数。综上可以通过检测输入来判断是否找到syscall，此时第二次调用read.got的三个参数就是1, 0x400000, 4。因为ELF文件首4个字节固定是\x7FELF。

通过这个方法搞到了syscall的地址。又已知execve的调用号是59，所以还是用read的返回值把rax设置成59，而syscall就直接顶掉read.got，三个参数分别是/bin/sh的首地址和两个NULL，即两个\x00。那么思路就清晰了，先第一次read，地址设置为read.got+59-1，设定读59个字符。这个地方写/bin/sh字符串，中间所有字符用\x00填充，最后一个字符就正好是read.got的最低8位（因为地址设置的时候还减了1），这里写入刚刚搞到的syscall低8位就行。然后三个参数，第一个就是read.got+58，即binsh地址，后两个就是binsh+8就行，因为那个位置已经用\x00填充了，所以值就是NULL。

* + - 1. **ret2VDSO**

粗略看了一下，跳过了，因为没讲原理，说的也比较基础和模糊。

* + - 1. **SROP**

操作不难理解，就是第一次可能稍微绕了一点。该方法主要利用sigreturn进行攻击，主要思路是伪造sigcontext已达到劫持程序的目的。

32位和64位系统的sigreturn都是通过系统中断实现的，前者的是int 80h，调用号77。后者用syscall或别的，调用号是15.

原理主要是两条：

① 由于signal frame保存在用户空间，且在栈上，所以用户可读写。

② 调用sigreturn时，他不会记录之前signal对应的signal frame，而是直接从栈上往下开始依次利用pop出上下文。

综上就可以在已有sigreturn时把对应的signal frame顶掉，或者自己造一个signal frame，然后通过系统调用主动调用sigreturn。rax的改变方法有很多，比较常见的就是利用read的返回值，只要read 15个字符（x64），返回时rax就是15 了，然后调用syscall就可以调用sigreturn了。

当我们伪造一个signal frame时，我们通常会控制rax，rdi，rsi，rdx，rsp和rip。rax是用来在后续syscall时指明目标函数的，最典型的就是通过写入constants.SYM\_execve来搞到shell。此时rdi，rsi，rdx就是三个参数。而rsp就可以写成我们希望栈指针跳转的地方，一般是我们控制的可写的地方。而rip就是下一条指令的地址，一般就指定到syscall的地址，这样当sigreturn完了就可以无缝衔接执行我们rax中指定的函数。最好的情况是syscall|ret的代码段，这样当目标函数执行完了，就可以直接ret到rsp指定的代码段，据此就可以无限延长我们的srop chain。当然，前提是空间够大。

本章给的例题比较简单，也很干脆。直接就是几行汇编代码，内容就是执行一个read，而且有意思的是，他read的前8个字符就直接覆盖了ret，我们甚至不用计算填充。

思路是把ret多次用程序起始地址覆盖（因为要反复利用main的代码段），为了得到栈地址（实际上是栈后面的地址），我们需要用write来泄露。调用write的调用号是1，所以我们只要通过read写一个1字符让rax变成1就可以实现。要注意的是，当rax变成1后，ret的地址要跳过rax清零的代码段，不然就白搞了。

当栈地址泄露后，就可以开始准备构造srop chain来执行execve了。但注意到源程序的rsi是直接用的rsp的值，所以我们不能直接利用他的read在栈地址处写入我们的代码。而且我们之后是要写入/bin/sh的，这个写入的位置只能是我们已知的获取的栈地址，当前栈的地址是不知道的。所以无论如果都需要伪造一个signal frame在栈地址处写入一串数据。

第一个signal frame自然就是让rax=constants.SYM\_read，rdi=0，rsi=stack\_addr，rdx=0x400（随便设的一个大小）。由于我们还要指定rsp，不然栈指针就飞了，所以当前所处的未知地址的栈就不能用了，rsp也得等于stack\_addr。rip就可以直接等于syscall|ret的代码段，当sigreturn执行完了就可以直接执行设置好的read。

在新栈位置设置栈结构为start\_addr + b’a’\*8 + sigframe+ 填充 + ‘/bin/sh\x00’即可，当跳过去后，需要再执行一次read把rax设置成15，此时顺便把8字节的a换成syscall|ret的地址，然后直接调用syscall执行execve（新的sigframe里rax=constants.SYM\_execve），rdi设置成binsh的地址，他的地址在已知填充的时候可以算出来，这样就A完了。

**4.2 format string**

**4.2.1 原理介绍**

该漏洞的存在主要基于两个条件：

1. 有一个缓冲区s可以让我们自由写入字符
2. 有一个格式化字符串输出函数（如printf），其中直接将s作为第一个参数（即格式化字符串）输出

此时我们可以通过往s里写入不同的参数来泄露栈内的信息。

调用printf后，x86栈内结构从低地址往高地址依次是。

ret->格式化字符串->参数1->参数2->参数3...

当为x64时，就按照顺序放到寄存器里。栈内顺序与正常调用函数时参数存放顺序是一样的。

常用的几个符号如下：

1. %x或%p来泄露栈内参数的值（内存），建议用%p，这样可以不用考虑位数的区别。
2. 利用%s来获取对应变量（内存单元）所指向地址的内容，但字符串会被\0截断。而且如果该内存单元里的指指向的不是有效地址，那就会程序崩溃。因此常用连续多个的%s来使得目标程序崩溃。
3. 利用%n$x来获取指定参数的值，其中n指的是第n+1个参数，这里的参数指的是栈内printf的参数。例如上面给出的例子栈结构中，实际上格式化字符串才是第一个参数，参数1才是第二个参数。但此处的n指代的就是参数1的后缀。因此类似的想要获取参数3的内容，那就是%3$x。而想要获取存有格式化字符串的内存单元的值理论上就应该是%0$x，但经过我的实践，证明了这个操作是行不通的，最后会原封不动的把%0$x打印出来。也就是printf认为%0$x就是个普通字符串，不需要读取任何额外参数。

**4.2.2 泄露任意地址内存**

虽然我们已经获得了泄露变量值的能力，但是最重要的还是要泄露内存地址。只有知道了地址才有很多操作可以进行。下面就介绍如何泄露任意内存地址。

由于printf要调用格式化字符串，也即我们写的缓冲区。在此之前缓冲区一定已经初始化好了并写好了数据，其一定是作为某个函数的局部变量（例如下例中的main函数）而存在栈中的。而后面调用printf时，也就是在栈上加了几个数据，因此后续的printf的第一个参数（即格式化字符串参数，实际上格式化字符串参数的值就是缓冲区的地址，但是我们不能直接搞到）和缓冲区的地址相差一定不远，而且缓冲区一定位于栈的下方，即高地址处。所以只要知道了缓冲区首地址与printf第一个参数的偏移，就可以通过%n$s来访问缓冲区里的数据。

为了找到这个偏移地址，我们可以这样设置我们的输入参数：  
 AAAA.%%%p%p%p%p%p%p%p....。后面的%p需要足够长以能读取到缓冲区的首地址中的值。

当printf把输出值打印出来后，只要找到打印结果中的AAAA，就能对比得到AAAA距离printf第一个参数的偏移，进而得到%n$s中的n。

当我们可以任意读取缓冲区中的值后，我们就能泄露任意地址内存。举个例子，假如我们已知write的got表地址，想知道里面的值，只要把格式化字符串设置为p32(write.got)+b’%n$s’。这样当输出函数会输出write.got的四字节地址(x86)后在输出write.got里的值，即write的真实地址（当然write要先被解析才行）。

要注意的是，这里使用的是printf(buf)，因此一定会先打印write.got的地址才会打印出里面的值，因为write.got写在字符串的前面。刚开始我实操的时候犯了傻，没搞明白为啥要跳过4字节，以为直接可以打印%n$s。后来才发现一时糊涂。

所以上述情况，got表里的值应该通过u32(sh.recv()[4:8])来获取，还要设置上限是因为%s遇到\0才会停下，不然会一直输出。

**4.2.3 覆盖内存**

意外的顺利，基本都是一次过，哪怕最后有个稍微复杂一点的函数编写，还是我稍微改变了一点的情况下。

覆盖内存主要用到%n，该符号标识将已经输出成功的字符数写入到对应的内存地址处。一次写入一个4字节的长度。如果想要一次写入一个字节就用%hhn，如果是两个字节就用%hn。

思路和正常泄露地址是一样的，首先先确定要被覆盖的地址，如某个变量的地址。这个方法很多，如果是初始化完成的全局变量，哪怕有ASLR也可以在data段里直接用IDA找到。但如果不是那就只能通过别的方法了，例如题中是直接用printf打印出了C的地址，简单粗暴。

然后是确定缓冲区首地址与printf第一个参数（格式化字符串）的相对偏移，两者相减除以机械字长（32位就4字节）即为%k$n的k值，即参数计数。

最后就是构造缓冲区，主要就是在适当的位置放上待覆盖地址，放上填充，在放上%k$n。待执行后即可写入。

题目中给出了三个覆盖的例子，分别涵盖了普通大小数字，小数字和大数字。前两者没有实质性区别，最后一个要特殊操作。

对于第一个，只要在buf首地址处写入待覆盖变量地址（题中为c），然后用待覆盖值减去待覆盖变量地址所占字节（32位就是4字节）就是需要填充的字节大小。填充上去后再写入一个%k$n就行了。这里k是缓冲区首地址相对printf的参数计数。

而第二个，如果是小数字，例如2，按照上面的流程是不可能实现的。因为带覆盖变量地址就占了4字节，也即写入地址无论如何都大于4字节。但实际上，待覆盖变量地址写入位置不一定非要是buf首地址，往后挪一下也是可以的，只是我们的k也要相应的变一下。例如要写入2，那payload就应该如下

b’aa%’ + bytes(‘{}’.format(count\_from\_buf\_to\_printf+2).encode()) + b‘$naa’ + p32(target\_addr)

因为要写入2，所以前面用两个a填充，然后紧跟着%k$n，这里写法比较复杂是因为payload得是byte类型，然而b’a’没有.format方法。所以要把count\_from\_buf\_to\_printf写入到字符串里后再encode（不encode会报错），然后转成bytes。这里count\_from\_buf\_to\_printf要+2是因为target\_addr在缓冲区首地址八字节后，除以机械字长4就是2，即count要+2。

上面操作完后就是b”aa%k$n”，但target\_addr要4字节对齐（32位），因为定位到target\_addr用的是k。而不是单纯地偏移，因此还要填充两个b’aa’。综上就得到了上面的payload。

可以看到，小数字和普通数字没有本质区别，只是target\_addr的位置变的随意了，唯一要多注意的就是要重新计算k以及target\_addr要内存对齐。不过不是难事。

有本质区别的是大数字，当时数字很大时，我们如果直接写入很大数字的填充八成会失败，这时就得把目标变量地址里4个字节（32位）单独拆开一个字节一个字节写。这里要注意的是计算机采用的是小端序还是大端，不过一般是小端序。

此时要用到%hhn，这样可以一个字节一个字节写，比较好使。但如果直接用%n也可以，但这样就一定要按顺序写最低字节到最高字节，否则如果先写高字节再写低字节，高字节的会被覆盖，而且最后还会被多覆盖3个字节（32位，64位则是7个字节）。可能引起错误，所以%hnn比较好。

此时payload结构类似下面:

addr1+addr2+....+addr4+pad1+%k$n+pad2+%k+1$n+pad3+%k+2$n+pad4+%k+4$n

即前面放目标变量的每个字节的地址（其实放在别的地方也可以，就是要算k）。后面就放填充和每个对应的'%k$n。这样的写法对大多数都可以，但有两个问题。

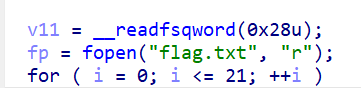
当某个字节的数字很小时，如果addr放在前面，那可能就会写不了，例如上面写法成立的前提是每个数字至少都要大于16（四个addr），这时就得把addr往后安排。

有可能某个地址本来写入的数字是可以满足的，但因为addr顺序安排不当导致满足不了。例如如果addr1里要写入0x80，而addr2里只用写入0x10。然而按顺序先写入到addr1（我习惯让addr和pad的顺序一致），那pad1就很大。addr2就写不进去。因此最好的办法在安排addr之前，先把四个地址按照四个字节要写入的数字大小先排序，让数字小的排前面，大的排后面。pad的顺序也随着addr变化依次变化，这样就可以一定程度上规避这个问题。我的实现中就先进行了排序然后才安排addr的位置。

**4.2.4 例子**

**4.2.4.1 goodluck**

本题题目本身没有任何难度，只要先找到buf和printf差异地址为5，然后由于是64位所以有6个参数在寄存器里。第一个参数是格式化字符串，因此%k$s的k选择10，之后直接sendline一个%10$s进去就完了。唯一的问题是这个程序会fopen一个默认为程序边上的flag.txt。由于那个fopen长这样



所以这个flag.txt要在程序执行路径下，因此出现了./c/goodluck段错误，但cd c再./goodluck却执行成功的傻卵问题。

**4.2.4.2 pwn3**

本题稍微难一点。但也还好。

首先这是个32位程序，其次这个程序在get\_file函数下有格式化字符串漏洞，程序在show\_dir函数最后调用了puts函数。所以就有了A掉的可能性。

先介绍本程序用的几个函数。

ask\_username要求输入一个用户名，并且会把输入的每个字符的ascii+1存储。而ask\_password就是检测刚刚输入的用户名等不等于sysbdmin字符串，但由于刚刚说道存储时每个字符asciii都+1了，因此密码就是sysbdmin字符串每个字符ascii-1组成的字符串，登录就完成了。

print\_prompt，没吊用，就是一个ftp的前缀输出。

get\_command，获取执行命令，输入get调用get\_file，输入put调用put\_file,输入dir调用show\_dir

put\_file，名义上是创建一个文件。实际上这个文件不写入文件系统，而是存在一个缓冲区。他的执行流程是这样的

1. malloc 244个字节，首地址为指针v1
2. 接受文件名，存在v1指向的地址，输入限制大小40字节
3. 接受文件内容，存在v1+10指向的地址，限制输入200字节
4. 在v1+60的位置存file\_head，即上个文件的缓冲区首地址
5. file\_head变成v1的值，即当前文件的首地址
6. 返回v1

这里容易误会的一点是v1+10并不意味着是在v1+10的地址处写文件内容，因为v1是\_DWORD\*类型，\_DWORD是4字节的。而这是C语言，也就是说v1+10应该实际上等于v1的地址加上40，所以文件名和文件内容不会重叠。同理，则v1+60则实际上是v1的地址加上240，因此是最后4个字节存储file\_head，所以认为每个文件是240字节，因为最后4字节是指向下个文件结构体的指针。我在这里卡了很久，一直没想到v1+10是要从C语言数组的角度结合类型来取地址的，导致我一度认为文件内容会覆盖文件名。

get\_file，存在格式化字符串漏洞的函数。

show\_dir，会调用puts打印文件名的函数。

综上，思路就清晰了

1. 找到printf时缓冲区与printf参数的距离，计算出%k$s中k的大小。
2. 通过elf得到puts的got，然后利用%k$s把puts的真实地址打印出来。（根据实践，got表地址不受alsr影响，因为这里是把alsr打开了的）
3. 通过puts的真实地址找到libc，算出偏移，得到system的真实地址
4. 再次通过格式化字符串漏洞，把puts的got表的值用system的真实地址顶掉，这里可以用pwntools的fmtstr\_payload，很好用，第一个参数给出缓冲区距离printf第一个参数的k，即%k$s的k。第二个参数是个字典，键是要写入值得地址，即puts.got的地址，值是这个地址里要写入的值，即system的真实地址。然后就弹出一个payload，把这个直接printf就可以ok了。
5. 要注意的是，第五步里同时还要把文件名设为/bin/sh;。不设置为/bin/sh\x00的原因是，在后面show\_dir里，输出缓冲区里字符是一个一个读进去的，当读到\x00时就直接跳出循环写下一个文件的名了。因此如果写入\x00会导致/bin/sh和之前的文件名连起来，如果之前文件名是1111，那就会形成/bin/sh1111的字符串，这样调用system会报错。而;在linux是让顺序从左到右按顺序执行的意思，所以会形成/bin/sh;1111，这样system会无视后面的1111先执行前面的/bin/sh，就没问题了。

⑥ 最后只要在执行一次show\_dir就行了，因为puts的got已经换成system，而根据反汇编观察，其参数s就是所有puts\_file的文件名，即⑤中的/bin/sh;1111，所以就等价于执行system(“/bin/sh;1111”)，执行成功。

本题中与解题无关但让我困惑很久的两点，这里记录一下：

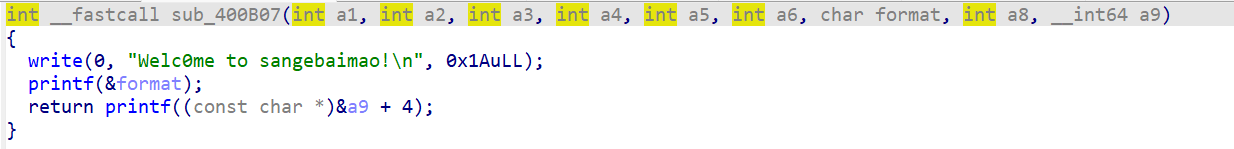
1. show\_dir中循环里i是+240而不是60的原因是，这里i是int类型，加240就是字面上数值+240，因此就是指教跳到文件结构体最后4位指向下个文件结构体的指针。
2. get\_file中循环里i+60是因为i首先被强制类型转换成了\*\_DWORD，然后才+的60，此时由于\_DWORD是4字节的，类似于结构体寻址，因此此时i+60等价于i的数值加60\*4。

**4.2.4.3 pwnme\_k0**

本题是在已经存在system(“/bin/sh”)基础上进行的，所以整体简单了不少。该题介绍的方法主要是ROP的方法，即将返回地址覆盖为system的执行地址。

该题的程序实现了一个用户注册的功能。初次运行程序会要求输入用户名和密码，用户名和密码都限制20个字节，这是算上\0的，即实际上最好只输入19个字节。同时用户名和密码在内存中的存储是连续的，即用户名首地址+20就是密码首地址。不过这个无关紧要（吐槽一下汇编里，这鸟程序居然是通过溢出的方法来存储用户名和密码的，因为开发者知道连续定义的变量内存是相邻的，但谁寄吧会这样写程序）

注册完成后有三个功能，输入1是展示注册的用户名和密码，就是在这个函数里存在格式化字符串漏洞。



可以发现，其实两个printf都有毛病，但ctf-wiki里反汇编的结果和我不太一样（但实际上也没差）。总之两个printf都可以用来操作，唯一的区别只是两个用的缓冲区首地址相差20个字节而已，任何需求都可以通过微调地址或填充来解决，这里就和ctf-wiki一样只用return的printf。

输入2后的功能是修改用户名和密码，没什么问题。3就直接退出了。

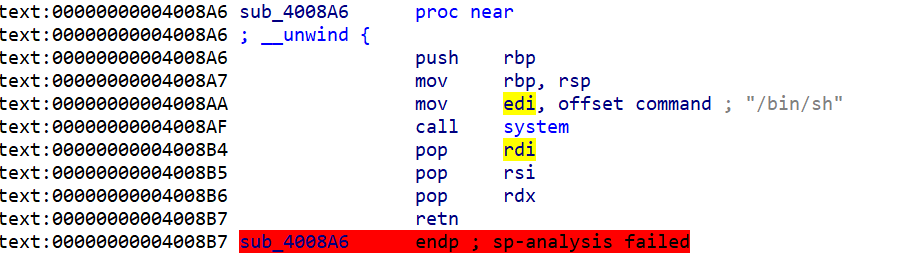
由于注意到开启了FULL RELRO和NX，所以不指望能改got表了，然而通过string发现居然程序里内置了/bin/sh，查看该字符串的被调用情况，就找到了一个现成的system。由于处于代码段，所以不用担心被alsr改地址。

之后的问题就是只要把函数返回地址覆盖成system的代码执行地址就行，但是我们不知道栈的地址，也因此不知道ret的地址。不过显然所有变量之间的相对位置是不会变的。我们通过观察栈结构，首先不难看出用户名的缓冲区地址相对printf首参数为8的偏移下标。并且缓冲区上面就是函数的返回地址，再上面就是rbp。所以6的偏移下标里存的就是rbp。而rbp的值和存有ret的栈单元的地址之间的相对位置是不变的。因此可以通过%6$p先提取出rbp的值，然后用gdb调试得到rbp的值和存ret的地址之间的差值。进而就可以泄露处存ret的栈地址。

然后就简单了，只要把这个地址写在缓冲区里，然后利用%k$n写入system的地址覆盖掉ret就行。我们同时注意到，system的地址和原ret的返回地址只有低2两个字节有区别，所以高位是不用变的。因此我们可以用%k$hn来修改，更加简便。

之前的例子已经知道了，%n是将先前成功输出的字符总数写入到对应的内存里。但存system代码段的低16位还是很大，怎么办？之前用的都是笨办法，一个字符一个字符的填充。但实际上我们可以用%2218d方便的解决这个问题，这样就会自动填充输出的数字到2218个。我也不知道前几题为啥不讲，可能是想让我深入理解把。

填充进去，在执行一次展示函数就OK了，不过要注意的是，这里低2字节可以写两个值。



注意到，我们可以直接把低2字节写成8AA,也可以是8A6。以前我们就知道system会有栈对齐的问题，所以可以先写在8AA，如果报错，表明没对齐16字节。此时在尝试8A6，先执行一次rbp会让rsp+8，那刚没对齐现在就对齐了。反之如果已经对齐了，那就不用试8A6了，八成出问题。

**4.2.4.4 contacts**

本题未经过我实际操作，只是看了题解，理解了主要解题思路。

本题程序主要实现了创建联系人信息，修改联系人信息与打印联系人信息等功能。其中在打印联系人信息中发现存在格式化字符串漏洞。

利用思路还是获取shell。但该题不能通过之前用过的修改got表或修改ret来获取shell。

前者是因为程序中常见的，可以对指定字符串输出的函数只有printf。但printf函数在程序别的地方也均有用到，所以改这个容易造成程序崩溃。

后者是因为没有现成的system代码段可以用来ret，而直接写入system\_addr+’bbbb’+binsh也不显示，毕竟只能用%n来写。

本题中创建联系人信息可以直接写入字符串，然而其写入的地址位于堆上而不是栈上。所以我们需要运用到栈迁移来把栈迁移到堆上。

在printf下断点，观察栈结构，发现三个可利用的点:（k等价于%k$p里的k）

1. 在k=6的位置存着一个地址，该地址下存着存有上个函数的ebp值的地址。即a->b->0，k=6的位置存着a，a存着b，b存着上个函数的ebp值0。
2. 在k=11的地方存着description（程序中的变量）的地址，该地址在堆中，且里面的值可以利用创建联系人信息任意写入。
3. 在k=31的地方存储的是main返回到\_\_libc\_start\_main的地址，观察这个地址所处的位置，将其值-241可以得到其首地址（实操的时候最好进到\_\_libc\_start\_main里面看看到底和首地址差多少）

又因为 PrintInfo 函数中存储 ebp 的地址每次都在变化，即①中存储a的地址总是在变，我们很难（除非保证函数调用每次申请和释放的栈空间一致）通过修改a来实现栈迁移。但我们可以通过修改b来解决这个问题。b之所以是固定的，主要是因为这是main函数返回时会pop 的ebp，且发现b指向的是0就可以知道b不可能变。这可以通过泄露a后，修改地址a的值来实现。

综上，可以得出利用思路如下

1. 通过%6$p，%11$p，%31$p把存ebp的地址(上文的a)，缓冲区的堆地址和\_\_libc\_start\_main的地址泄露出来。
2. 根据刚泄露出的\_\_libc\_start\_main地址，推测首地址，进而找到libc，然后获取system的/bin/sh的地址。
3. 构造基本联系人的description为system\_addr + 'bbbb' + binsh\_addr
4. 通过修改地址a存的值（利用%k），即利用修改上层函数ebp（上文的b）为缓冲区堆地址-4（栈迁移利用的leava指令中有个pop，-4可以正好让ret指向system\_addr）来实现栈迁移。
5. 返回程序以栈迁移并get shell

**4.2.4.5 blind\_fmt\_got**

本题是盲打劫持got，没有经过我的具体实践，此处仅放我的个人理解。

首先可以通过输入一个%p发现存在格式化字符串漏洞，并可以发现程序是64位。

然后通过输入aaaaaaaa%p%p%p...可以得出缓冲区距离printf首参数的距离。

然后通过%k$s循环来dump出从0x400000开始的整个程序。这里要注意的是，如果字符串里有\n，即\x0a，则源程序只会读取这个前面的内容，而没法泄露内存，因此只能忍痛割爱，当地址出现这个字符时就直接把这个地址的值替换成\xff。

dump出程序后对程序分析，发现有个read函数和输出函数，并且在while 1反复执行。因此思路就是泄露printf的地址，然后获取libc和system地址，把printf的got改成system的地址，读入/bin/sh后获取shell。

后面有个要注意的是fmtstr\_payload 直接得到的 payload 会将地址放在前面，而这个会导致 printf 的时候 '\x00' 截断（关于这一问题，pwntools 目前正在开发 fmt\_payload 的加强版，估计快开发出来了）。所以ctf-wiki使用了一些技巧将它放在后面了。主要的思想是，将地址放在后面 8 字节对齐的地方，并对 payload 中的偏移进行修改。

**4.3 heap exploitation**

**4.3.1 堆溢出**

堆溢出是用户输入字节数超过堆块本身可使用的字节数（不一定等于用户申请的字节数，例如64位系统，malloc申请8字节，但由于对齐要求16字节，所以会给该chunk分配16字节数据段，还要加上下一个chunk的prev\_size，所以一共可以用24字节）。

一般，堆溢出的前提有两个：

1. 程序往堆上写数据。
2. 没有控制写入数据的大小。

与栈溢出不同的是，这里不能控制ret等可直接控制程序执行流程的参数，因此无法通过堆溢出控制RIP（EIP）。一般堆溢出的策略如下：

1. 覆盖与其物理相邻（这里的物理是虚拟地址的物理）的下一个chunk的内容。主要是下一个chunk的prev\_size，size（里面堆的大小和三个比特位），chunk\_content，从而改变程序固有的执行流程。
2. 利用堆中的机制（如unlink）来实现任意地址写入或控制堆块中的内容等效果，从而控制程序的执行流。

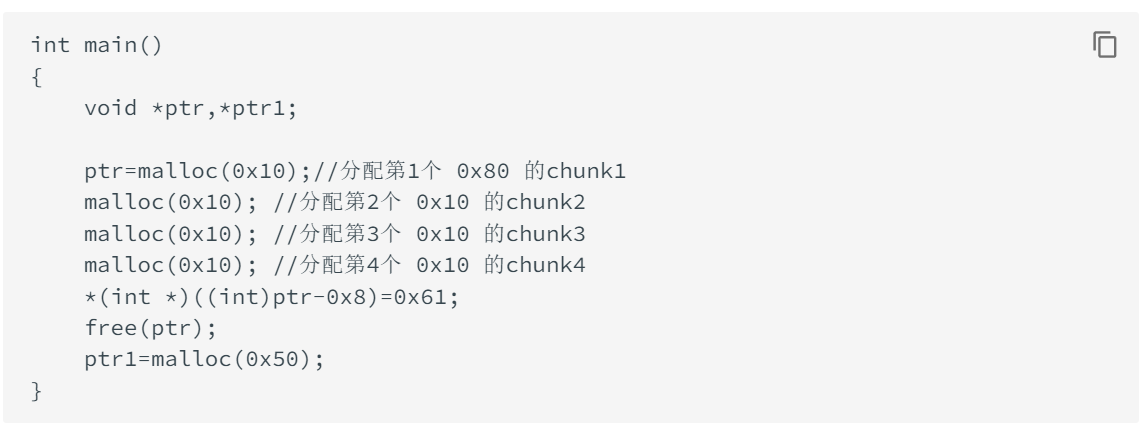
有几个要注意的点如下：

1. calloc函数会对分配的堆自动清零，等价于malloc后memset内存为0。这使得部分信息泄露漏洞不可实行。
2. realloc可以身兼malloc和free两个函数。且realloc有几个特性
   1. 当realloc的size大于原堆的size，如果chunk和top chunk相邻，则直接扩展chunk。否则会先free掉原来的chunk再malloc新的size
   2. 当realloc的size小于原堆的size，如果相差容得下一个最小的chunk（64位是32字节，32位是16字节），那就切割原chunk为两部分，并且free掉后一部分（应该是高字节）。如果容不下一个最小的chunk，那就保持不变。
   3. 当realloc的size等于0，等于free掉原堆。
   4. 如果realloc的size和原size相等，什么操作也不做。
3. 一定要确定填充长度，这和请求长度不同。例如请求24字节，实际上会给数据段16字节（即整个chunk size 32字节），因为还有8字节可以直接利用下一个chunk的prev\_size。同理如果请求16字节，那虽然确实数据段是分配了16字节，但实际上下一个chunk的8字节也可以用。

**4.3.2 Chunk Extend and Overlapping**

chunk extend是一种常见的利用手法，主要是通过extend以实现overlapping的效果。当漏洞可以控制chunk header中数据时，就可以考虑该方法。

由于连续申请的小堆块在地址上是相连的，所以如果控制了某一个堆块首部的size，则可以将该size设置的足够大使得覆盖到下面几个（高地址）的堆块首部和内容，进而起到控制下面几个堆块的效果。例如下面这个程序



根据堆的数据结构，可以知道ptr的前8个字节就是堆头部的size变量。将其修改为0x61后，则ptr所在的堆块内容实质上还包括下面的chunk2和chunk3，但不包括chunk4，因为chunk2和chunk3的大小都是0x20（包含头部）。

当free ptr后，0x60大小（包含头部）的堆块进入了fast bin。然后再malloc就又搞到了该堆块，ptr1指向其content。此时就可以通过修改ptr1来任意控制chunk2和chunk3。

上面演示的是控制高地址的堆块，但实际上还可以控制低地址（上面）的堆块，这主要是通过修改size的pre\_inuse来实现的。见下面这个例子程序、



128就是0x80，因此ptr1和ptr4的size都是0x91。当free了ptr1时，那个堆块会进入small bin（fast bin最大只能存0x80大小）。此时修改ptr4的size为0x90，即将pre\_inuse 标志位置0，表面前一个堆块未在使用。并且将pre\_size修改成0xd0，因为ptr1的堆块+ptr2的堆块+ptr3的堆块大小等于0xd0。再次free ptr4，就会将ptr4的前一个堆块认成ptr1的那个堆块，就会触发合并。此时再次申请该堆块就可以控制ptr2和ptr3指向的堆块。

**4.3.2.1 HITCON Trainging lab13**

本题的程序功能是创建自定义堆，修改自定义堆，展示自定义堆的内容，删除自定义堆。

在修改自定义堆的函数里存在off by one漏洞。但溢出的one不是\x00，是我们可以自己修改的。

由于原程序的自定义堆实际上是个结构体，两个成员，第一个是size\_t大小的变量size，描述content指向的缓冲区大小。而第二个成员自然就是指针content，在创建自定义堆时会先malloc这个结构体，然后根据size的大小malloc出content指向的堆。因此每个自定义堆都是由两个堆组成，位于低地址的堆存着该结构体，高地址的堆存着content的缓冲区，且两者地址上是连续的。

因此可以调用创建自定义堆申请缓冲区0x18的自定义堆a，然后申请缓冲区0x10的自定义堆b。此时0x18的堆块的size实际上是0x20，因为借用了下一个堆块的pre\_size。此时修改0x18的堆块内容为"/bin/sh\x00" + "a" \* 0x10 + "\x41"，既可以为/bin/sh做准备，又可以让溢出的\x41正好覆盖到下一个堆块的size，而这下一个堆块就是刚创建的缓冲区大小0x10的堆块。

调用删除自定义堆函数时，会先free掉content，然后free掉该自定义堆结构体。此时调用删除自定义堆b的函数。content被正常释放，放在bin大小为0x20的fast bin里。然后结构体堆被释放，放在bin大小为0x40大小的结构体堆里。

之后再次调用创建自定义堆，申请缓冲区0x30的自定义堆c。由于结构体堆大小只有0x20，因此指向原来堆b的content首地址，而由于堆c的缓冲区大小为0x30，所以要0x40的堆。所以新缓冲区content指向原堆b的结构体堆首地址。这一操作使得修改自定义堆c的content正好可以操作自定义堆c的结构体。即两者结构如下。

heap header1(size 0x40)->space size 0x10 ->heap header2(size 0x10) -> size -> ptr content

即content指针指向space size 0x10，自定义堆结构体指针指向size。现在只要利用修改自定义堆函数把ptr content用free函数（用这个函数是因为之后要调用）的got表地址覆盖。之后调用展示自定义堆，就可以泄露free的真实地址。

通过该真实地址确认libc，找到相对偏差，获取system真实地址。

由于got表可写，再次调用修改自定义堆函数，这时写的地址就是got表的地址。把system的真实地址写上去，就把free的真实地址替换成了system的真实地址。之后调用删除自定义堆函数删除自定义堆a。那就会调用free(content)，等于调用system(content)，自定义堆a的content里写了/bin/sh\x00，所以get了shell。

**4.3.3 Unlink**

**4.3.3.2 2014 HITCON stkof**

首先声明，该题未能复现出结果，问题出在unlink没能触发，原因不明。但依然可以叙述解题思路。

该题目实现了三个功能，alloc，read\_in, free。

alloc是申请一个堆，并把返回的堆的content指针存放于bss段的变量s中。（即s是个数组，里面存着alloc申请的指针）。alloc申请的第一个堆下标为1，s[0]为0。因为他的计数变量count是先自增然后存放，即下方代码：



其中::标志s是全局变量，dword\_602100就是计数器count，v2是刚申请堆返回的指针。

read\_in在输入堆指针，输入字节大小和输入字符串后就可以往堆里写数据。由于没有限制输入字节大小，因此这里存在堆溢出。

free就是单纯地释放目标堆指针。

由于本题未setvbuf，所以首次调用fgets和printf时都会malloc分配缓冲区。fgets没办法，肯定是先执行的，所以让他分配。但printf的首次执行在alloc里，那里面是先申请堆，之后printf一下堆的下标。因此堆的结构是fgets\_chunk->ptr1\_chunk->printf\_chunk->top chunk。

为了让堆可以溢出到我们到我们可以控制的区域，我们再次调用alloc申请两个堆，第一个大小为0x20（ptr2 size 0x30），第二个大小0x80（ptr3 size 0x90）。这两个chunk是连在一起的，因为不会再给printf malloc缓冲区了。

由于unlink利用的效果是让某堆指针p=&p-0x18。我们知道在bss段里，ptr2指向的并不是chunk head，而是其content。所以为了满足unlink的条件，我们在content处重新构造一个fake chunk。其结构为：

p64(0)+p64(20)+p64(&ptr0+16-0x18)+p64(&ptr0+16-0x10)

fake chunk的prev\_size无所谓，所以设0。其大小是0x20（不确定prev\_inuse设0有没有影响，可能设1最好）。后面两个变量是fd和bk。&ptr0可以通过IDA在bss段找到（alsr应该不影响bss段地址）。&ptr0+16就是&ptr2。根据公式，fd里要-0x18，而bk要-0x10。

在构造fake\_chunk的基础上，继续往payload增加p64(0x20)+p64(0x90)，覆盖ptr3 chunk的prev\_size为0x20，指向我们的fake chunk，并把其size的prev\_inuse标志位设为0，标志该fake chunk未使用。随后free ptr3，则理论上当free ptr3时，会对fake chunk触发unlink。此时又因为我们构造好的fd和bk可以通过检测（根据ptr3-prev\_size 会寻址到ptr2 chunk的content首地址ptr2，而ptr2->fd->bk=ptr2->bk->fd=ptr2），所以ptr2会被更改为&ptr2-0x18，即&ptr0+16-0x18。（然而就是在实践中这一步失败了，原因不明）

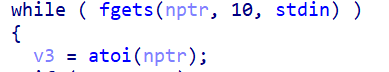
此时可以通过read\_in函数，以ptr2为参数来修改ptr0，ptr1，ptr2。因为ptr2本来的含义是ptr2 chunk的content，但现在指向了&ptr0-8。构造payload为b'a'\*8+p64(elf.got["free"])+p64(elf.got["puts"])+p64(elf.got["atoi"])，使得ptr0指向free got，ptr1指向puts got，ptr2指向atoi got。

现在需要泄露puts的地址，因此使用read\_in修改ptr0，把free的got改成puts.plt（因为puts之前没运行过），这样free ptr1时就等于puts(ptr1)，就泄露出了puts的真实地址。

通过puts真实地址泄露出libc，搞到偏移，获取system和binsh的真实地址。

上面说道，ptr2又被覆盖成了atoi got，因此再次read\_in修改ptr2，就可以把atoi的got替换成system的真实地址。

由于整个程序在一个大循环里，每次循环开头都会fget一个输入，然后调用atoi。所以我们此时把binsh的地址send到fget里，之后调用atoi就等价于system(binsh)，则成功。

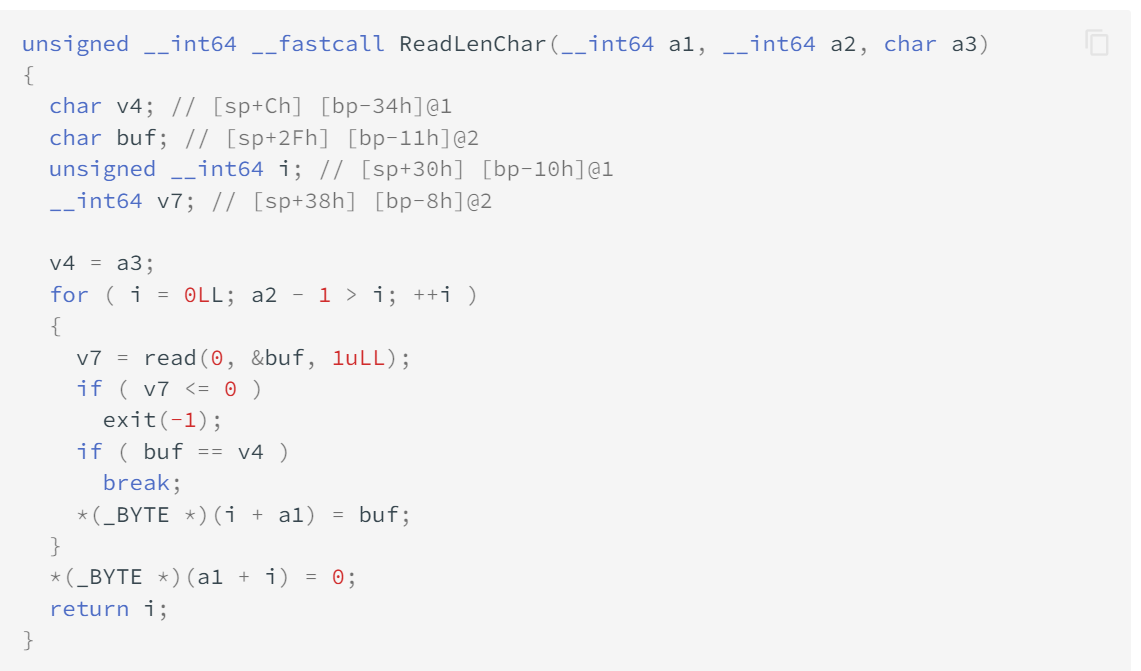


**4.3.3.3 2016 ZCTF note2**

本题依然没能复现成功，原因还是unlink没能触发，但理论和思路可以叙述。

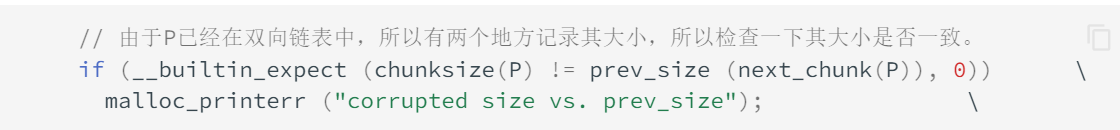
功能是添加note，展示note内容，编辑已有note内容（覆盖或添加），删除note。

其漏洞有二个，一个是当我们输入note大小为0时，由于glibc根据规定会分配0x20大小的字节。且该大小会被记录，用于控制读取note的内容。然而，读取时用的循环变量i是无符号整数，所以比较时都会转换成无符号整数。则当note的size为0时，又正好因为-1转换成无符号数是最大的整数，所以可以近乎无限的输入数据。下面代码的a2就是read的size，当为0时-1就是最大的数。



第二个漏洞是每次编辑note时，都会申请0xa0大小的内存，但free后没设为NULL，但这个没用上。

漏洞还是主要利用unlink来把ptr转到&ptr-0x18。触发方式是先申请一个chunk，然后再他的content里伪造fake chunk。设置fd为&ptr-0x18，bk为&ptr-0x10。然后为了满足以下条件。



在content+size的地址处写上size大小。unlink的检测理论上就过了。

之后在申请一个0 size的chunk2，系统会分配0x20的size，再随便申请一个chunk3。之后free掉chunk2，在申请回来，这次申请回来的时候申请的size还是0，这样就可以无限写数据。我们把chunk3的prev\_size指向fake chunk头，size中的prev\_inuse标志位设为0。准备工作就做完了。

free掉chunk3，这时会触发fake chunk的unlink，理论上ptr就会变成&ptr-0x18。然而这次实践还是失败了，原因不明。

之后通过修改note把atoi（用这个是因为每次循环开头都用了这个函数）的got地址写入到ptr里，执行显示note泄露处atoi的真实地址。找到libc，找到偏差，找到system地址。

然后通过修改note把atoi的got表的值改成system真实地址，输入/bin/sh，get shell。

**4.3.4 use after free**

终于复现成功了一次，本题漏洞主要是指针free后没有将指针指向NULL，使得释放后也依然可以调用指针指向的函数。

Use After Free主要是以下两种：

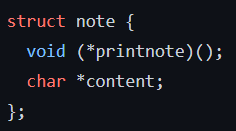
① 内存块被释放后，其对应的指针没有被设置为 NULL ，然后在它下一次被使用之前，没有代码对这块内存块进行修改，那么程序很有可能可以正常运转。

② 内存块被释放后，其对应的指针没有被设置为 NULL，但是在它下一次使用之前，有代码对这块内存进行了修改，那么当程序再次使用这块内存时，就很有可能会出现奇怪的问题。

我们一般称被释放后没有被设置为 NULL 的内存指针为 dangling pointer。

**4.3.4.1 lab 10 hacknote**

该程序有三个功能，且定义了一个struct，struct结构如下图。



第一个是一个指向函数的指针，第二个是指向数据存储区的指针。

该程序是32位的，因此最小的chunk大小为16字节。

程序三个功能，分别是添加note，打印note的content，删除note。每次添加note后都会将返回的note指针存放于一个数组处，且删除note也只是单纯地free指针，没有将指针指向null。因此哪怕note删除了，也依然可以通过该数组访问指针下数据。即可以再输入索引后通过print\_note函数调用note的printnote成员。

程序中有一个magic函数，效果是system(“cat flag”)。通过gdb搞到这个函数的首地址。

首先申请两个note，note0和note1大小都是16字节。然后再把两个note删除掉，这样就搞到了两个16字节的fast chunk，两个24字节的fast chunk。且两个16字节的fast chunk的在fast bin里顺序是note1->note0。因为先释放note0再释放note1。

此时在申请一个note3，申请大小8字节，则note1正好指向note3的结构体，note0正好指向note3的content首地址。

在申请note3的同时，把content的前4字节设置成magic函数首地址。

然后调用print\_note，索引为0。此时note0的函数指针的值就等于magic函数首地址。调用完成后就搞到了flag。

**4.3.5 fastbin attack**

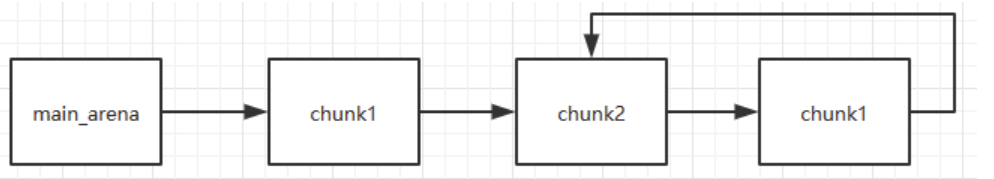
**4.3.5.1 fastbin double free**

由于小size的chunk，在被free时，下一chunk的prev\_used还是1。所以该chunk被操作系统识别为仍在使用中。所以如果该chunk被free后指针还不设置为null，那就可以被double free使得该chunk被两次加入到fast bin中。

为了防止double free，fast bin再加入chunk前会检测当前链表头的chunk和即将被free的chunk是不是同一个chunk，如果是就报错。但这个检测很没用，因为只要free chunk1，free chunk2，再free chunk1就可以绕过这个检测。

该漏洞可以实现多个指针指向同一个堆块，结合堆块的内容可以实现类型混淆（type confused）的效果。不过下面的演示主要还是利用该漏洞实现任意地址写，该利用核心是利用fd。因此可以简化，见4.3.5.3 alloc to stack。

通过合理的利用该漏洞，可以实现在任意位置分配chunk。例如我们还是先free chunk1，free chunk2,再free chunk3。不妨设三个chunk都是0x20大小则此时fast bin里如下。



此时假设我们能控制chunk1的内容。例如在C语言里我们又malloc 0x10。那么就控制了chunk1。此时返回的指针ptr指向chunk 1的fd指针。因此我们直接把ptr指向的8个字节（64位）替换成目标地址。

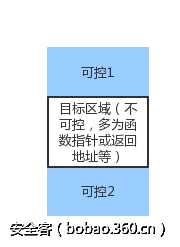
注意目标地址的size（即64位情况下目标地址+8的地址指代的变量）在这个情况下必须是0x21，因为fast bin分配前会验证目标chunk的size是否与其所处的bin相等。

然后再malloc 0x10搞到chunk2，再malloc 0x10又搞到chunk1。此时由于chunk1的fd指向我们构造的目标地址，因此fast bin以为还有一个chunk在链表中。main\_arena就指向了这个目标地址。最后再malloc 0x10就实现了在目标地址分配chunk，这等价于任意地址写。

**4.3.5.2 House Of Spirit**

可以看这篇文章，讲的更详细一点。（https://www.anquanke.com/post/id/85357）。

这个技术主要应用于以下这种情况。



即目标区域我们不可控，但目标区域上面和下面都是可控的。这时为了控制目标区域，我们可以考虑使用这种技术来控制目标区域。方法就是伪造fast chunk。

在可控的区域，我们可以伪造好size，使得正好能覆盖全目标区域。该size大小不能超过fast bin的最大大小（x64就是0x80）。所以如果目标区域大于0x80，那肯定就不行了。且该size的ismmap位不能为1，即第二位不能为1。

要注意的是，fake chunk的地址还需要对齐MALLOC\_ALIGN\_MASK，即0x10。

对于可控2的区域，也有一定的要求。fake chunk的next chunk的size不能小于2\*SIZE\_SZ（x64下就是0x10），不能大于system\_mem。因此在下一个chunk的size地址（通过刚刚size的地址加size的值求出）随便设置一个合适的就行。

之后只要找个指针指向可控1的size地址-8的地址，然后把这个指针free掉，fast bin里就多了这个chunk，之后再malloc回来就能控制目标区域。

**4.3.5.3 alloc to stack**

从fast bin double free中我们可以观察到，实现在任意地址分配堆的关键在于fast bin的一个特性，即当前chunk的fd指向下一个chunk。

因此理论上，只要我们能够劫持fast bin中chunk的fd，我们就能实现任意地址分配。同理就可以实现在栈上分配堆。

在fastbin double free中，为了演示double free，我们当时申请了三个堆。但事实上，如果第一个堆free时没有设置null，我们就能实现任意地址写。因为第一个堆free时没设置null，我们就能直接设置该堆的fd指向栈。栈的地址可以通过获取局部变量地址来取得。然后再把这个堆malloc出去，下一次malloc就是栈上的堆。

然而，这种方法需要提前确保确保栈上对应size的数据在0x20到0x80之间，这可能需要我们手动调整或者通过泄露的方式寻找合适的数据。而且在确定好这个size后，我们才能知道前面第一个堆的size应该申请多少。因为两个理应相等，否则会报错，原因在fastbin double free中提到过，fastbin在分配前会检测待分配的chunk是否与其所处的bin理应存的size相等。

**4.3.5.4 Arbitrary alloc**

实际上就是alloc to stack的通用版本。实现任意地址分配。事实上，只要能找到合适的size（size是构造的还是自然存在的都无所谓，在4.3.5.1中的例子在ctf-wiki里就是构造的）。

假设我们有构造size的能力，那可以首先考虑house of spirite方法，但那个方法是以free个假chunk，比较麻烦。

假设没有构造的能力，这样意味着想要在目标地址分配chunk就需要在目标地址前面找一个合适的size。例如ctf-wiki中想要控制\_malloc\_hook，他就在\_malloc\_hook地址前找一个合适的自然存在的值0x000000000000007f作为size，是否合适的标准就是值是否在0x20到0x80之间。

由于标准的size都是0x10对齐的，所以在通过size获取fast bin的index的代码中，是直接把size右移4位（等价于忽略低4位）然后减2得到的index。因此上述size的index就是5，对应的fast bin中可以存储的size就是0x60，这里提到的fast bin中可以存储的size是指数据区的大小，即实际上分配出来chunk的size是0x70。

知道size的值和地址后，就可以malloc 0x60先在idx=5的fastbin里占个位置，之后free掉这个chunk。然后假设我们可以通过某种手法（例如free后指针没置NULL）来调整fd，那就把fd的值设置成刚刚找到的size的地址-8（因为chunk header起始地址就是&size-8），然后再malloc 0x60把这个中间chunk排出去。下一次malloc 0x60就可以控制目标地址了。

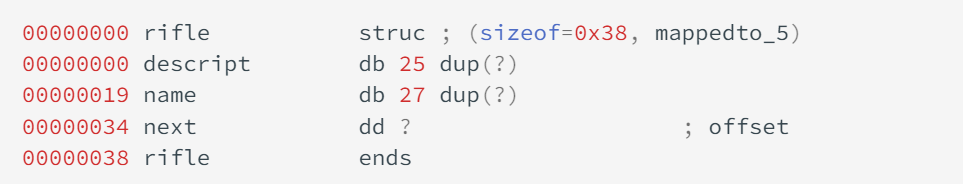
这个方法用的比较多，且自然生成的size一般就是通过字节错位找到的。

**4.3.5.5 hack.lu oreo**

此题未能复现成功，在free fake chunk的地方程序崩溃，原因暂时不明。但可叙述思路。

本程序是32位程序，实现了枪支购买相关功能。主要是添加枪支，展示添加枪支，购买，留下订货信息，展示目前状态。

每个枪支由一个结构体维护，其形状如下。



这个结构体是0x38字节，因此其申请的chunk size是0x40字节。（32位chunk header只有8字节）

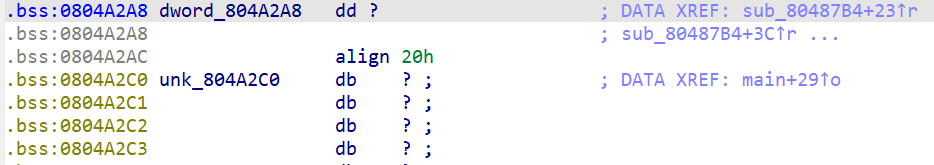
在添加枪支时，会先设置next指针，然后添加descript，最后是name。同时在name处可以写56个字节，此处存在堆溢出。可以覆盖next指针，也可覆盖下一个堆。每个新添加的枪支都处于枪支链表的链表头。

展示添加枪支就是顺着枪支链表依次输出des和name。

购买，实际上是顺着枪支链表依次free掉枪支信息结构体。

留下订货信息是在一个缓冲区里写入相关数据。值得一提的是，该缓冲区与缓冲区指针之间距离不远。见下图。



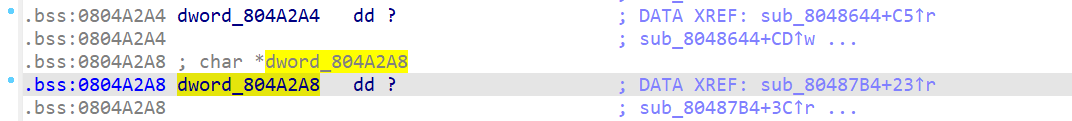


本题利用思路是，首先添加一个枪支结构体，在填写name时溢出next，把next用puts.got覆盖。然而展示添加枪支，进而获得puts真实地址，进而获取libc，偏差，system真实地址。

然后思路是控制dword\_804A2A8，因为这个变量存着订货信息缓冲区的地址。假如我们能把这个变量的值替换成任意地址，然后再调用“留下订货信息”就能实现任意地址写。

为了控制这个变量，我们使用刚学的House of spirite方法。没有用任意地址写是因为不能控制fastbin中的fd。

为了实现这个条件，观察dword\_804A2A8前4个字节。



在程序中找了一下，发现这个dword\_804A2A4存着添加枪支的数量。而这个位置是我们伪造chunk的size的位置。

观察发现只有购买函数里调用了free，因此想要能够free 0804A2A8。我们必须把0804A2A8也放在枪支链表里，因此需要实现下述结构。

枪支结构体->0804A2A8->NULL

为了实现上面这个结构，枪支结构体的next值就得通过溢出设置成0804A2A8。而0804A2A8的next得设置成p32(b‘\x00’)

并且为了通过检测，fake chunk的size设置成0x40(因为之后只能通过添加枪支获取chunk，而一个枪支结构体是0x40大小)。因此需要先申请0x3f个枪支结构体，最后一个枪支结构体里的next存着0804A2A8就行。

为了搞到system的地址，已经申请了一个枪支结构体，因此只要循环申请0x3e个枪就可以了。

申请完了后，观察0x0804A2A8+52的地址，发现正好就落在信息缓冲区里。因此就好办了，只要在那个地址设置成0。这个fake chunk就伪造完了。

与此同时，为了通过检测还需要在fake chunk下面伪造next chunk size，这个在伪造fake chunk时顺便写一个数据就行。

全部伪造完成后，调用购买函数。理论上会free首部的枪支结构体后free 0804A2A8。此时这个fake chunk就进入了fastbin的0x40的bin里。然而实践里再free fake chunk这一步崩溃了，原因不明。

假设成功了，接下来只要再次申请枪支，fake chunk就又弹出来了。我们在申请枪支时把descript的前4字节写成p32(elf.got[‘strlen’])，其他的随意。选择strlen是因为之后还要调用留下订货信息，当订货信息写完后会调用一个函数，里面调用了strlen。

此时订货信息缓冲区的地址已经被改了。选择留下订货信息，把订货信息写成p32(system\_addr)+’;/bin/sh\x00’即可。/bin/sh写在这里是因为后哦面调用strlen时参数就是订货信息缓冲区的地址。所以调用strlen等价于调用下面函数

system(p32(system\_addr)+’;/bin/sh\x00’)

在linux里，;可以让命令从左到右执行，不关心是否执行成功。因此上面会首先执行system(p32(system\_addr))，肯定失败了。然后就会接着执行system(“/bin/sh\x00”)，这次肯定成功了。

**4.3.6 unsorted bin attack**

**4.3.6.1 基本原理**

unsorted bin attack被利用的前提是可以控制unsorted bin chunk中的bk指针。其最终效果是可以实现修改任意地址为一个较大的数值。

unsorted bin有三个来源：

1. 一个较大chunk被分成两部分，剩下的部分大于MINSIZE，该部分会被放到unsorted bin chunk
2. 释放一个不属于fast bin的chunk，且该chunk不和top chunk相邻。该chunk会首先被放到unsorted bin中。
3. 进行malloc\_consolidate时，可能会把合并后的chunk放在unsorted bin中，如果chunk不与top chunk相邻。

unsorted bin的使用情况如下：

1. unsorted bin不同于fast bin，是采用先进先出的顺序。插入时也会插入到unsorted bin头部，但取却是从末尾取。由于unsorted bin是双向链表，因此取末尾也比较简单。
2. 程序malloc时，如果在fast bin和small bin中找不到对应大小的chunk，就会试着从unsorted bin中找。如果大小刚好满足就直接返回给用户，否则会把这些chunk插入到对应的bin中。

通常main\_arena的地址比\_\_malloc\_hook要大0x10。由于大多数libc都可以直接查到\_\_malloc\_hook的地址，所以main\_arena在libc中的地址可以通过右式得到：main\_arena\_offset = ELF("libc.so.6").symbols["\_\_malloc\_hook"] + 0x10

攻击原理见ctf-wiki中的解释：

<https://ctf-wiki.org/pwn/linux/user-mode/heap/ptmalloc2/unsorted-bin-attack/>

其主要流程如下：  
 ① 搞到一个堆a（大小要大于fast bin的max大小0x80）

② 把这个堆free掉

③ 将a->bk修改成target\_addr - 0x10（即把target\_addr视为fake chunk的fd）

④ 把堆a在申请回来

⑤ 申请回来时，会进行一些操作（实质上就是双向链表最后一个节点脱离时会让倒数第二个节点的fd指向头）使得target\_addr下的值指向bin的首地址，又因为这个地址很大，且不能为我们控制。所以说会让target\_addr下的值变成一个很大的值。

**4.3.6.2 magicheap**

复现失败，但思路很简单。失败原因是我当前版本的glibc已经基本杀死了unsorted bin attack。在这篇文章中提到了这一点：

<https://blog.51cto.com/u_15127704/4005164>

这同时也让我意识到，之前的unlink，fake chunk free等利用失败八成也是这个原因，由于加入了许多严苛的检查使得都失败了。不过这一点还是我的猜测，没有找到论据，不过八九不离十。

总之还是先说说题目，题目实现了三个功能，申请堆，编辑堆，释放堆。在编辑堆时出现了堆溢出漏洞。

这一题中存在一个函数是我们最终要调用的（可以搞到flag），但由于调用他前他会验证一个全局变量magic是否大于4869。但这个全局变量初始化是0。所以实现思路就是利用unsorted bin attack将这个变量的值变的很大就OK了。

由于关闭了pie，所以可以直接在bss找到magic变量的地址。之后只要随便申请一个堆，然后再申请一个大于fastbin大小的堆（但实际上在当前版本还要大于0x512，因为如果小于等于small bin的size则被free后会首先被放到Tcache bin里）。之后随便申请一个堆防止释放大堆时和top chunk合并继续。然后把中间的大堆给free掉，他就掉进unsorted bin里了。然后通过溢出修改第0个堆，把第1个堆（大堆）的bk给改成&magic-0x10。然后再把大堆给申请回来。这时magic的值理论上就变了（但因为glibc检查严格了，所以这里申请回来时程序就崩溃了）。

最后直接调用目标函数就行，magic已经够大了所以肯定过的了检测。

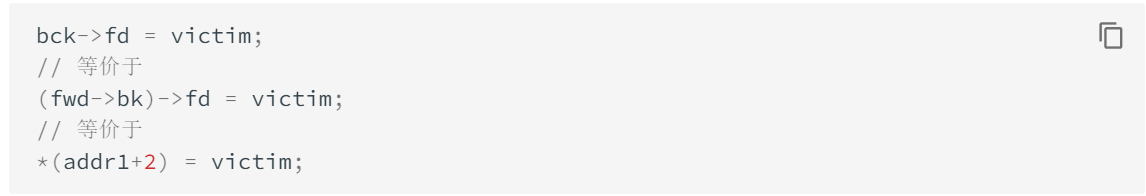
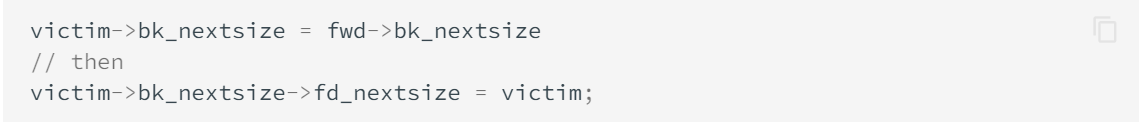
**4.3.7 large bin attack**

该攻击方法同样是修改任意地址的值，但其修改的值不能任意指定。不过依然可视作为某个较大的值。该值实质上是之后申请的某个堆的地址。

想要触发该攻击，首先需要能够修改一个large bin chunk的data段。最好还包括chunk header段。如果不能包括chunk header段，则要求从unsorted bin中来的large bin chunk可以紧跟着被构造的chunk前面（比构造chunk的size大）。否则则必须通过修改被构造chunk的size实现这一点。

该攻击可以同时修改两个地址&a，&b的值。需要做的主要操作如下（默认为x64，且忽略chunk free时的合并问题）

1. 搞到一个large bin的chunk1，并设法把他free到large bin中。
2. 修改该chunk的bk为&a-0x10，bk\_nextsize为&b-0x20。同时如果有必要，修改size使得接下来的chunk2的size正好比他大，且在large bin中正好与他相连（即正常流程chunk1->bk\_nextsize=chunk2）
3. free掉chunk2，并设法让他掉进large bin中。此时会因为我们尽精心构造的chunk1的size使得chunk2正好掉在chunk1的前面，此时根据链表中间的的插入方法，自然会修改插入位置前后链表的指针。因此触发了以下操作



上图中victim为chunk2，fwd为chunk1。

当上面的操作发生时，chunk1->bk\_nextsize已经是&b-0x20了，故第一个图的两个代码执行后，就会让b的值变成chunk2的首地址。

同理，chunk1->bk已经是&a-0x10了，故第二个图的代码执行后，就会让b的值变成chunk2的首地址。

上面已经讲解了漏洞的基本触发方法，但是我们可以注意到，第一步和第三部中都free了large chunk。但事实上，直接free chunk1和chunk2是不会让他们直接掉进large bin的。而是先会掉进unsorted bin。等到某次malloc后触发unsorted bin的chunk整理机制才会把unsorted bin中的chunk放到对应的bin中。因此在ctf-wiki中才会在一开始申请一个small size的chunk，即其给出的例子中的指针p1。

为了让chunk1掉进large bin，他先同时free了small chunk和chunk1，两个掉进了unsorted bin（顺带一提，顺序是chunk1<->small chunk）。然后再malloc 0x90。在分配前首先会把chunk1和small chunk放回各自该有的bin中，此时就完成了把chunk1放回large bin的工作。当然，之后会从small chunk中扒一块0x100 size大小的chunk下来。并且剩余的chunk因为大于MINSIZE所以就又放回unsorted bin中了。之后就是类似的操作把chunk2也放回large bin就行了。

**4.3.8 Tcache attack**

**4.3.8.1 tcache poison**

比较类似于fastbin attack的arbitrary alloc。但是fastbin的检查比较多样且花里胡哨，而tcache就几乎没什么检查，所以实现起来就容易的多。

tcache投毒只要简单地把一个chunk的fd改成目标地址就行。当然，这要求我们能够控制一个被free的chunk的data段，这是前提。

ctf-wiki中给的例子的步骤概括如下：

1. 申请一个chunk（大小要小于large chunk）
2. free掉这个chunk
3. 把这个chunk的fd修改成目标地址
4. malloc一次，由于tcache是单向链表，因此是LIFO。刚刚free的chunk被弹出去了
5. 再malloc一次，这次malloc的值当然和上次是一样的，因为挂载在上面那个大小的tcache bin下面。而这次malloc就把目标地址当成chunk弹出来了。

**4.3.8.2 tcache dep**

类似于fastbin double free。然而fastbin对于double free还会检测本次添加的chunk是否是上次添加的chunk，即是否与当前bin的头部指向的地址相同，相同还会报错。但tcache什么检测都没有，所以double free就格外方便。直接申请后连着free同一个指针两次就行了。之后只要连着malloc两次就能获得两个指向同一地址的指针。

**4.3.8.3 tcache perthread corruption**

tcache\_perthread\_struct是整个tcache的管理结构，如果可以控制这个结构那么无论malloc的size是多少，地址都是可控的（ctf-wiki原话，但还没理解为什么）。

想要控制这个部分的地址有很多方法，其本质就是如何利用任意写控制这部分地址。最简单地方法就是刚刚学过的tcache poison。直接申请个堆过来，想怎么写怎么写。

tcache\_perthread\_struct里面有个counts数组和一个entries数组。个人猜想如果控制了这个数据结构，可以通过控制counts数组里的值使得无限制的增加（或限制）某个entry链表里的chunk数量。或是直接构造一个fake chunk直接塞到entries里某个bin里。又甚至直接打印出entries里那些chunk的首地址等。

注意，由于tcache\_perthread\_struct也在堆上，所以只要实现partial overwrite 就可以达到目的。

**4.3.8.4 tcache house of spirit**

fastbin attack中的house of spirit的简化版。实质是一样的，也是实现指定地址分配chunk。然而由于tcache简陋的检测机制，只要fake chunk的prev\_inuse和size是合理的就行。不过也要注意IS\_MMAPPED和NON\_MAIN\_ARENA需要是0。

house of spirit中还要在fake chunk中设置一个fake prev\_size，使得可以通过检测。然而tcache版本的house of spirit连这个都可以省了。只要设置好size就行。

因此利用流程如下：

1. 随便malloc一个值，先把堆区初始化了。
2. 找到需要设置控制的区域a，在&a-8的位置，即size的位置设置好我们的值，值不能大于0x408，即不能到large bin的size。
3. 找个指针指向&a-0x10，然后free掉这个指针
4. 之后只要malloc size就可以控制这片区域了。

**4.3.8.5 smallbin unlink**

smallbin中有空闲堆时，当执行malloc时会顺便把small bin里的空闲堆扔到tcache里。此时也会有解链操作，但相比unlink的解链操作缺少了链的完整性校验。因此原本unlink操作在该条件下也可以使用。

**4.3.8.6 tcache stashing unlink attack**

该攻击方法最后可以在任意地址分配fake chunk，并且在任意（可写）的地址写上bin（一个libc）的地址。

其核心原理在于当small bin里有chunk，而tcache对应的bin里chunk没满时，如果调用calloc，则calloc不会从tcache里取数据，反而是会从small bin里取数据。

在调用calloc取该chunk时，会进行一次链完整性校验（\_\_glibc\_unlikely (bck->fd != victim)），然而只会进行这一次，之后把small bin中空闲chunk搬运时是不会校验的。

当chunk取完后会顺便把small bin里的其余空闲chunk按顺序（small bin里是FIFO，所以从尾部开始取）把chunk依次搬运到tcache里。这个过程中的解链不是unlink，但很类似，不过没有完整性校验，所以很松。

当把空闲chunk依次移到tcache里时，每个chunk解链操作都肯定有这两个步骤：bin->bk = bck;bck->fd = bin;。因为是从尾部抽走chunk，所以bin->bk肯定要指向新的尾部chunk，故有bin-bk=bck。而新的尾部chunk，即bck的fd就要指向头bin了，所以bck->fd=bin。

观察上面的代码，可以发现bck->fd=bin是给待unlink的chunk的前一个chunk的fd进行赋值。因此只要控制了待unlink的chunk的bk，就可以实现任意地址写libc地址（一个比较大的值）。

上面解释了为什么可以在任意地址写一个比较大的值（一个libc的地址），下面解释怎么分配fake chunk，同时顺便演示整个漏洞利用的过程

首先需要说一下初始条件，也可以理解为初始环境：

假设我们在small bin中有两个chunk并且在small bin中结构如下：  
 bin->chunk1->chunk2

同时我们才tcache中对应的bin里有随便几个chunk，例如两个，结构如下：

entries[n]->chunk3.data->chunk4.data

下面开始叙述漏洞触发流程：

此时为了让漏洞触发成功，我们必须通过某种手法（例如tcache poison或UAF）修改chunk1->bk的值为我们想分配chunk的地址，假设该地址为&a，则chunk1->bk=&a。

那么显然，我们的意思就是把&a当一个fake chunk的首地址分配出去。

此时我们还需要让fake chunk的fd值为一个可写的地址，例如&b，即a->fd=&b。

那么显然，当我们calloc之后，chunk2会被分配出去，此时会进行一次检测\_\_glibc\_unlikely (bck->fd != victim)。而chunk1因为tcache中对应的bin，即entries[n]未满，所以从尾部开始解链，并送到entries[n]上面去，送的时候是利用bck不断循环的，因此chunk1搬完后不会直接结束，而是因为chunk1->bck=&a而再次把&a也当个chunk的首地址搬运过去。因此整个流程结束后entries[n]中是这样的：

entries[n]->&a+0x10->chunk1.data->chunk3.data->chunk4.data

entries[n]的第一个元素是&a+0x10是因为tcache里fd指针指向的都是下一个chunk的data段，而不是header。

显然，在把chunk1解链时，会进行bin->bk = bck;bck->fd = bin;。所以当到这一步时,bck->fd=chunk1->bk->fd=a->fd=b。因此bck->fd=bin等价于b=bin。故此时b的值就变成了一个libc的值。

同时，如果再次进行malloc，就会把&a+0x10作为一个chunk分配出去。

综上所述，我们只需要完成以下步骤（ctf-wiki里演示步骤）：

1. malloc九个大小相等的chunk，size<large bin size
2. free掉下标为3-9的chunk（送进tcache）
3. free chunk1，free chunk 0，free chunk2（chunk1送进tcache，剩下两个送进unsorted bin。不让unsorted bin中两个地址连起来免得两个chunk合并了）
4. malloc一个不同的size（与上面9个不等即可，触发unsorted bin送进small bin的操作）。
5. malloc两个相同的size（给tcache里空两个位置）
6. 修改chunk2->bk=&a，修改a->fd为&b（理由见上面的解析）
7. calloc一个相同的size（送走chunk0，并把chunk2和fake chunk送进tcache，与此会同时修改b=bin）
8. malloc一个相同的size可以把&a-0x10当chunk弹出来，且此时b=bin

**4.3.8.7 leak libc**

泄露libc地址有助于我们后续的找到system等操作。在以往没有tcache时泄露libc比较简单。只要申请一个比较大（大于fastbin）的chunk，这样当free掉该chunk时。他就会掉进unsorted bin中。由于unsorted bin是个双向链表。且此时该bin中只有一个chunk，所以fd就指向bin，bin就位于libc中。因此就泄露出了libc的地址。

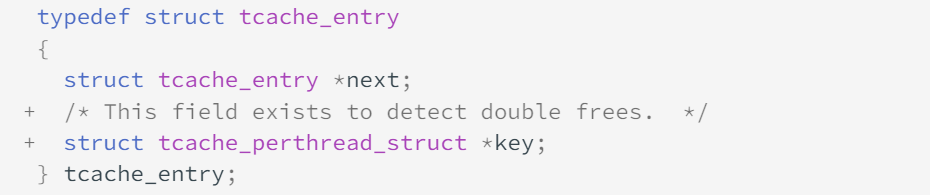
然而有了tcache后，小于等于small size的chunk都会优先掉进tcache bin中。虽然large chunk不受影响。但毕竟还是稍微麻烦了一点。

对于large chunk而言，还是free后直接打印他的fd就行。

但是非large chunk的话就得先申请7个一样大小的chunk，然后free掉填满tcache bin。随后再申请一个size一样的chunk（之前一块申请好也行），free掉就掉进unsorted bin中了。这时打印出他的fd就泄露出libc了。

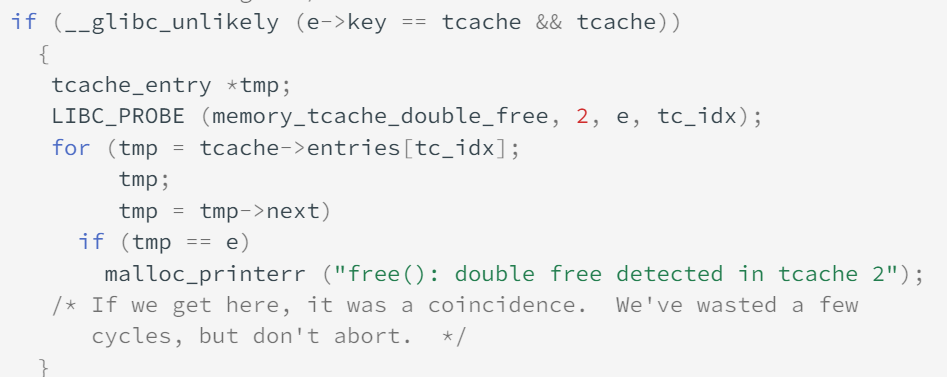
**4.3.8.8 Tcache check**

目前已经更新了针对Tcache的double free检测。同时tcache\_entry略有变动，见下图：



注意到多了一个key变量。观察源代码后发现所有挂载在tcache bin上的chunk的key都指向了最初的tcache变量，这个变量里就存着count和entries。

double free check的代码如下图所示：



其中，e就是待free的chunk。注意到，进入第一个if判断时，会检测e->key是否等于tcache。显然如果e已经添加过了，那么e->key肯定就是等于tcache的。但由于e可能没添加进去，e->key是tcache也可能是巧合，所以会进一步遍历链表看是否已经添加过了。

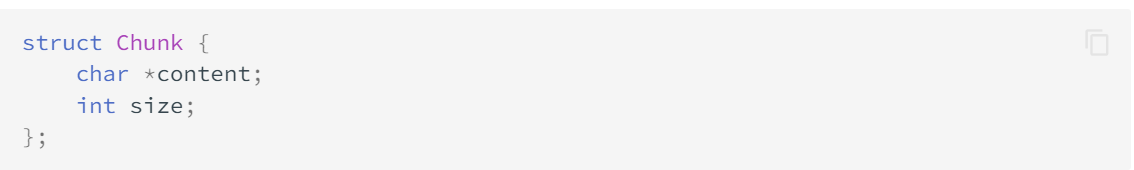
**4.3.8.9 LCTF2018 PWN easy\_heap**

一个用到tcache的大题，然而因为刚刚收到实验室那边的项目，所以可能这是短期内最后一次学习pwn了，之后估计要先学学reverse相关的知识。

由于该题目中用到了tcache 的double free，然而在我的环境里tcache check已经上线了，所以我都不用试就知道我复现不出来。不过由于该题用到了不少知识，这里还是总结一下操作过程。

该程序为64位，full relro，canary found，nx enabled，pie enabled。基本上是防护全开，改不了got也难以栈溢出。地址还被混淆完了，这意味着我们只能用相对地址来搞事。

有个记录chunk的结构体，程序初始情况时会重复申请十个指针，都是struct Chunk类型。即该程序最多new十个chunk。



函数功能如下：

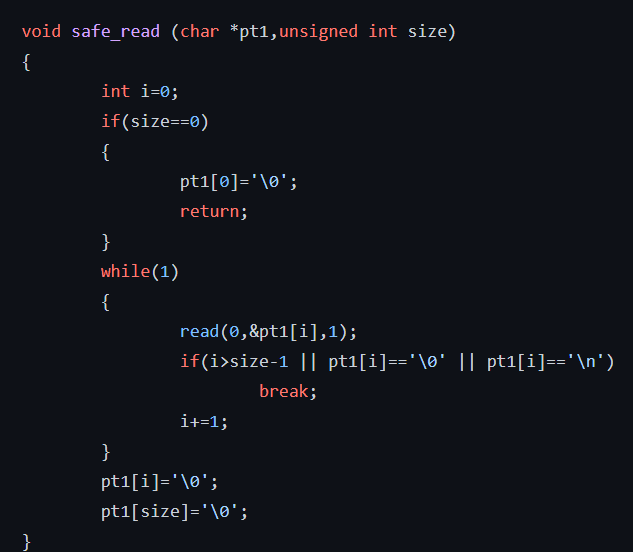
new：使用malloc(0xa8)给content新建一个堆，并且要求输入size，最大0xa8。

free：根据记录下的size将content给free掉，并且会把指针和content内容都置零。

show：使用puts输出contents

输入函数：循环读入一个字节，出现null字节或换行符就停止输入。并且最后会对读入的末尾位置和size位置置零。这里存在off-by-NULL漏洞。

注意到每次new的content大小都是0xa8，这表明整个chunk大小为0x100，且会用到下一个chunk的prev\_size。由于输入函数存在off-by-NULL漏洞。因此假如我们new的时候size设置为0xa8，则下一个chunk的prev\_inuse会被置零（因为会在size位置置零，见下图pt1[size]=’\0’）。



见到这个漏洞，意味着我们得用到chunk extend and overlapping。

由于这个版本的glibc还存在tcache，因此后续操作还需要考虑到他的存在。

利用chunk overlapping还需要我们能够控制prev\_size。但是该程序没有编辑chunk的功能。所以我们不能直接操作prev\_size。但是注意到当unsorted bin中chunk合并时会修改prev\_size。所以我们的思路是巧妙的操作unsorted bin中的chunk使得prev\_size变成我们需要的数值。

能够操作的空间上面叙述完了，下面叙述我们需要怎么做才能搞到shell。

首先我们需要搞到libc基地址，只有搞到基地址我们才能用相对地址获得one\_gadget（execve(‘/bin/sh\x00’,0,0)）的地址。进而把这个地址覆盖到\_\_free\_hook中。

为了搞到基地址，我们需要把bin泄露出来，泄露bin的方法就是当unsorted bin中只有一个chunk时，该chunk的fd会指向bin。

之后n号chunk，n按地址大小排列，0号chunk地址最小，9号chunk地址最大。如果说1号chunk3，则表明该chunk地址第二小，但是在程序的chunk数组中下标为3。简称1chunk3。

因此我们先执行以下步骤：

1. new 10个chunk。
2. free 0~5号chunk，free 9号chunk，free 6~8号chunk（0~5+9号chunk填充tcache，6~8号chunk进unsorted bin。9号进tcache避免进unsorted bin的chunk也和top chunk合并了）
3. 上面操作后，6~8号chunk就合并成了一个大chunk，此时8chunk8-0x10=0x200。prev\_size设置好了

**下面准备溢出7chunk使得8chunk的prev\_inuse为0。**

1. new 7个chunk，清空了tcache。这些chunk分别为9chunk0, 5chunk1, 4chunk2, 3chunk3, 2chunk4, 1chunk5, 0chunk6。
2. new 3个chunk，从unsorted bin中取。会把大的6号chunk又拆分三个chunk出来。所以得到6chunk7, 7chunk8, 8chunk9。
3. 为了让8chunk被溢出，我们需要把7chunk放到tcache里，这样可以释放后又分配出来。因此free chunk0~5，然后free chunk8，7chunk进tcache。
4. 然后free 6chunk7，6chunk进入unsorted bin中。
5. new 1个chunk，7chunk从tcache里弹出来了，作为7chunk0，此时设置size为0xf8，因此8chunk的prev\_inuse被设置为null。

**现在我们准备要free 8chunk9使得合并成一个大chunk**

1. 然而此时tcache不满。因此我们首先执行free 0chunk6，然后执行free 8chunk9。此时6chunk会合并掉8chunk，即6chunk的size变成0x300。

**下面我们要泄露处libc地址**

1. 现在已经存在一个7chunk0。而unsorted bin中有一个6chunk，且6chunk包含7chunk和8chunk。故new 7个chunk，排空tcache，得到0chunk1，1chunk2，2chunk3，3chunk4，4chunk5, 5chunk6，9chunk7。
2. 再new一个chunk，得到6chunk8。由于unsorted bin中的大6chunk被拆了一块，因此头部变成7chunk，故7chunk->fd=bin。而7chunk0已经存在了。
3. 展示7chunk0的内容，泄露bin地址。
4. 去libc里面查出bin距离libc头部的相对地址，得到libc真实地址。

**现在已经有了libc地址，可以找到one\_gadget的真实地址，下面我们要通过double free把one\_gadget写到free\_hook中。**

1. new一个chunk，搞到7chunk9。
2. free 0chunk1和1chunk2，送进tcache里，便于后面绕过tcache的检查。（绕过什么检查还不知道，之前学的检查是搬运small bin时的检查）
3. free 7chunk0和7chunk9。构成了double chunk。因此在tcache里构成了下面的链：bin->7chunk.fd->7chunk.fd。即7chunk的fd指向了自己。
4. new一个chunk得到7chunk0，并在此时写入\_\_free\_hook的真实地址。使得tcache里变成了这样bin->7chunk.fd->\_\_free\_hook
5. new一个chunk得到7chunk1。tcache里变成这样bin->\_\_free\_hook
6. new一个chunk得到free\_hook的chunk，并在此时写入one\_gadget的真实地址。
7. free 7chunk1（随便释放一个），触发free\_hook，触发one\_gadget，即execve(‘/bin/sh\x00’,0,0)。拿到shell。

上面的过程比较抽象，主要是数组编号和地址实际大小没有对应。最好还是看ctf-wiki和博客（https://blog.csdn.net/qq\_41202237/article/details/113697892），结合起来理解。两者都有图，虽然在搞libc时中途做法有一点差异，但区别不大。我的总结是基于ctf-wiki来的。

1. **基础知识**
   1. **GOT和PIL**

为了实现动态链接，当文件编译链接后会形成GOT和PIL表。由于大的前提，操作系统不允许修改代码段，只能修改数据段，因此为了实现动态链接必须在数据段里写好待链接的函数地址。

其中PIL表位于代码段因此不可修改。链接时链接器会特地为地址不明的函数（在动态链接库里的函数）在代码段写入一个小代码段（可以理解为接口），用于事后间接寻址跳转到真实函数地址。GOT表里就记录了函数的真实地址，PIL表会直接调用GOT表里的数据进行跳转。PIL表中执行的实际过程类似下图



（https://linyt.blog.csdn.net/article/details/51636753）

当需要执行printf函数时，会call printf@plt，进入到上图的函数。Printf@got就是记录printf函数真实地址的got表地址。由于延迟重定位的特性，got表一开始没有把所有需要重定位的函数真实地址都写好。而是在里面记录了jmp \*printf@got这条指令的下一条指令的地址（其他的指令也是类似的），即lookup\_printf的地址。所以首次执行时jmp \*printf@got会直接跳转该指令的下一条指令。此时会调用专门的重定位函数（\_dl\_runtime\_resolve）查找printf函数的地址，并且该定位函数定位完成后会顺便把真实地址填写到printf@got里，即写入到printf的got表里。然后回到printf@plt的起点再执行一次，这次printf@got里因为记录的是printf的真实地址，所以就正常跳转过去了。

（https://www.zhihu.com/question/21249496/answer/2057605287）

* 1. **\_dl\_runtime\_resolve**

资料主要来源ctf-wiki和深入理解-dl\_runtime\_resolve的博客（https://ctf-wiki.org/executable/elf/structure/basic-info/#section-header-table）（https://blog.51cto.com/u\_15127656/4538661）

**补充，下面两个博客里写了解析的详细过程（https://blog.csdn.net/jzc020121/article/details/116312592）（https://pythontechworld.com/article/detail/iAD6PiSHmozZ）**

动态链接器在解析符号地址时所使用的重定位表项、动态符号表、动态字符串表都是从目标文件中的动态节 .dynamic 索引得到的。

当进行符号解析时，主要涉及到以下几个节:.plt, .got, .rel.plt, .dynsym, .dynstr。根据我目前学到的内容来看，解析不区分变量和函数，统称符号。

符号解析主要通过调用函数\_dl\_runtime\_resolve来实现。其结构如下：

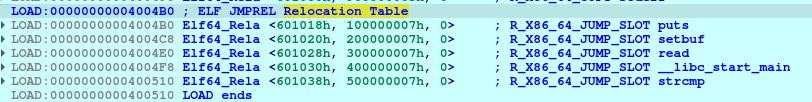
\_dl\_runtime\_resolve(link\_map\_obj, reloc\_offset)

有一点比较奇怪的是，尽管这是64位程序，但该函数依然是通过栈传递参数，后面可以看到这两个参数从右往左都是push进去的，等日后明白了再回来解释。

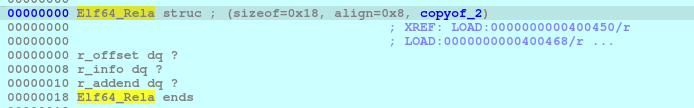
第一个参数用于获取解析导入函数所需的信息，第二个参数reloc\_index则标识了解析哪一个导入函数，这里的reloc\_offset等于对象在.rel.plt中的偏移（64位是下标），例如puts在里面就是第一个元素，所以reloc\_offset是0。当延迟绑定时，在plt里会先跳到got再跳回来，这时会push一个值，这个值就是reloc\_offset

解析涉及到的两个重要数据结构如下。

1. 根据reloc\_offset可以定位到Relocation Table中的目标元素。这个表形状入下图。

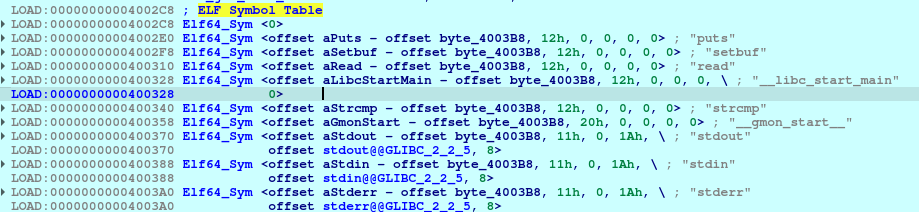


ELF64\_Rela结构体内定义如下图

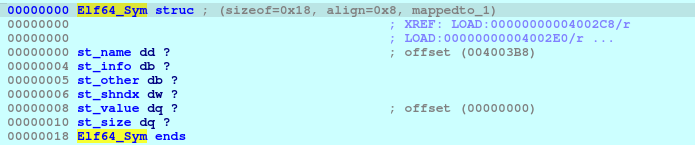


第三个成员不管，32位里没有，应该也不是很重要，可以看到运行时值都是0。R\_offset的值等于该符号在got表里的地址，严谨一点即该变量用于保存解析完成后的符号地址写入内存的位置（绝对地址）。同时按照博客中的说法，R\_info的高位三个字节用于标识该符号在.dynsym中的下标。但根据我的实际测试，应该是四字节，可能是因为64位的原因，和32位有区别。可以看到上面图中puts的r\_info就是1 00 00 00 07h。然而其下标理应是1，这可以在后面的展示中验证。

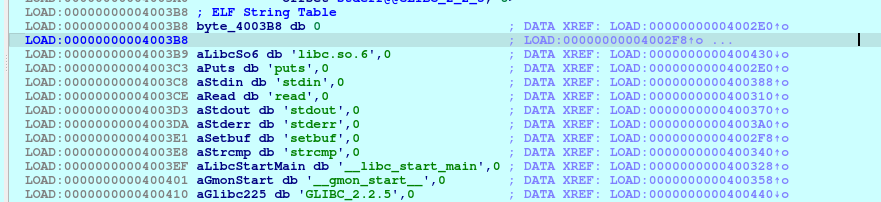
1. 根据r\_info定位到ELF Symbol Table中的目标元素，这个表形状如下图。



可以很明显的看到，1号元素就是puts。0号元素是全零占位符。所以上述关于r\_info的结论应该是正确的。ELF64\_Sym结构体的定义如下图。



St\_name是该符号在字符串表中的索引，或者说偏移。由于字符串不是定长存储的，所以不能用下标来跳转，这个是比较**重要的一个成员**。字符串表结构如下图



St\_info给出符号的类型和绑定属性。St\_other含义未定义，应当为0。

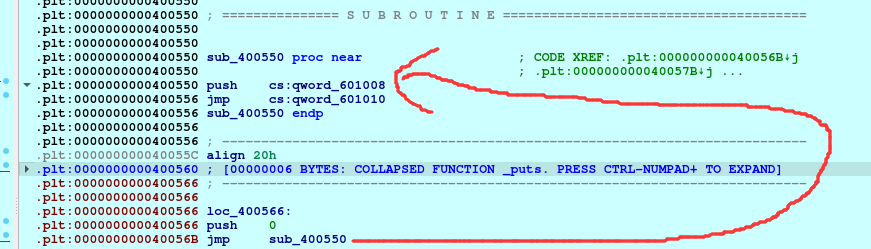
St\_shndx要分情况讨论。如果该符号定义在该文件中，则该成员为符号所在节在节区头部表的下标，可以用来溯源。如果不在，对于某些特殊符号，他具有特殊含义。

St\_value也是一个**比较重要的成员**，当该符号被导出时，该成员保存着对应的虚拟地址。

St\_size给出符号占用的大小，如果没有大小或大小未知则为0。

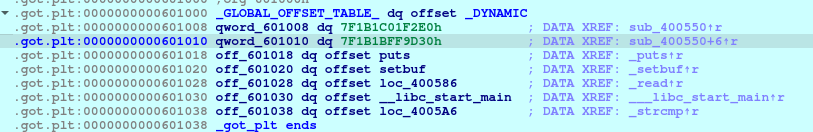
总结一下，当程序导入函数时，动态连接器先在.dynstr段加一个函数名称字符串。然后再.dynsym段添加一个指向函数名称字符串的ELF\_Sym结构体。然后在.rel.plt段添加一个指向ELF\_Sym的ELF\_Rela结构体。最后根据ELF\_Rel中r\_offset形成GOT表，保存在.got.plt中。

当开始延迟绑定时，首先从PLT跳到GOT，再跳回plt的下一条指令，即push。此时push的是待解析的函数在.rel.plt中的编号（下标），即reloc\_offset。然后就会jmp到进入\_dl\_runtime\_resolve前的一个小代码段，如下。

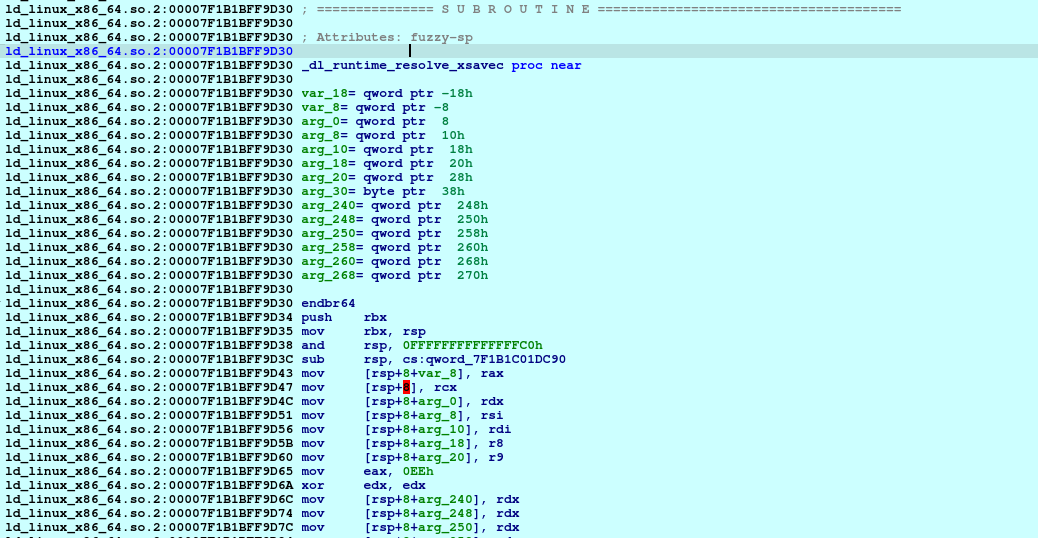


这里先push了一个参数，大胆猜测应该就是link\_map\_obj。即第一个参数。然后就jmp到了\_dl\_runtime\_resolve的代码执行片段。

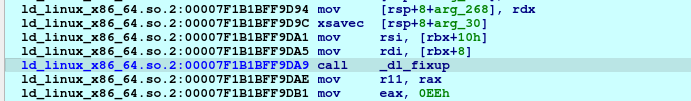
因为之前搞错过所以顺带一提，这里push和jmp的是变量的值，即下图中7F1B1C01F2E0h和7F1B1BFF9D30h。而不是601008和601010本身。这是个很愚蠢的问题，但之前脑子糊了就是没转过来。



跳过去之后可以看到已经进入了函数的代码执行段。从这里总体来看，尽管是64位程序，但依然采用的是比较原始的栈传递参数的方式。



该函数中调用一个叫\_dl\_fixup的函数，该函数用于解析导入函数的真实地址，并改写GOT：



* 1. **.dynamic**

这鸟表给我整麻了，我得记录一下部分项目的实际内容。

DT\_PLTREL：值是DT\_REL或DT\_RELA的d\_tag。

DT\_PLTRELSZ：ELF JMPREL Relocation Table的表大小。

DT\_JMPREL：ELF JMPREL Relocation Table的首地址

DT\_RELA：ELF RELA Relocation Table的首地址

DT\_RELASZ：ELF RELA Relocation Table的表大小

DT\_RELAENT：ELF RELA Relocation Table的单元大小

DT\_PLTGOT：got.plt的首地址

ELF RELA Relocation Table和ELF JMPREL Relocation Table里头的值是Elf64\_Rela的值。结构长这样。5.2有详细介绍



R\_offset里存的对应函数got或got.plt表项的地址。

**5.4 堆**

**5.4.1 堆介绍**

程序运行过程中，堆可以提供动态分配的内存，运行程序申请大小未知的内存。由低地址往高地址增长。管理堆的程序叫堆管理器。当用户释放内存时，内存不会直接还给操作系统，而是由堆管理器进行管理。一般堆就在bss段下面（关闭alsr）或下面一段偏移处（打开alsr）。并且程序申请字节很小时，一般会直接给一块很大的内存，避免内核态和用户态多次切换，提高了程序的效率。

**5.4.3 关于(s)brk和mmap**

当主线程调用malloc等方法分配内存时，通常使用sbrk往下推动edata。

当线程分配内存时，通常利用mmap在edata和栈中间挑个位置分配内存。

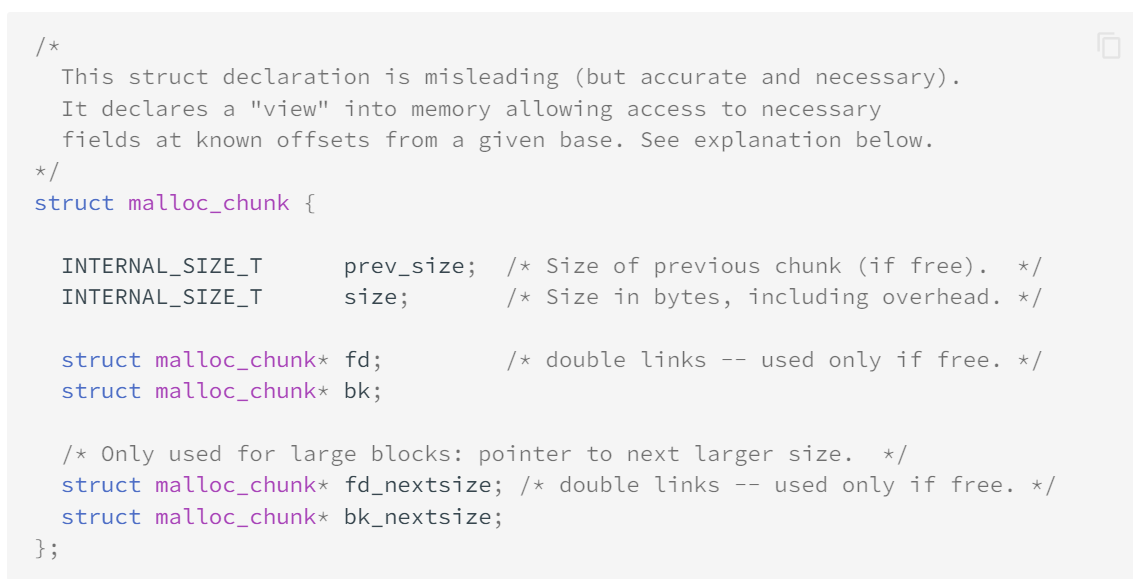
当分配的内存大小大于128KB时，如果没有空闲的空间可以分配，那无论请求来自哪个线程，都使用mmap分配内存。

利用sbrk往下推的话，如果堆结构类似A->B->C，回收B的空间时，那么edata不能往回退，除非C也被回收了。但如果利用mmap，那么其中任意一个内存空间想单独释放就单独释放。

具体见博客:https://blog.csdn.net/shuzishij/article/details/86574927

**5.4.2 堆相关数据结构**

堆的数据结构比较简单，且无论每个chunk是否被分配，都使用统一的数据结构。数据结构如下图所示，通常称呼前两个字段（变量）为chunk header，后面的叫user data。每次malloc申请得到的内存指针指向user data的起始处，即fd变量的地址。在chunk中存在大量的空间复用，见下文的解释。



prev\_size： 如果某个chunk物理相邻的前一个chunk未被分配，则该变量存着前一个chunk的大小（包括chunk头）。如果前一个chunk被分配了，该变量所处的内存空间会被前一个chunk征用，用来存储数据。

size：存储该chunk的大小，包括chunk头。大小必须是size\_sz\*2的整数倍。其中size\_sz是INTERNAL\_SIZE\_T的大小（64位是8字节，32位4字节）。并且该变量的最低3位不表示chunk的大小，而是征用来标识几个标志位（反之因为大小必须是size\_sz\*2的整数倍，不征用这里一定是0）。这几个标志位从高到低分别是

1. NON\_MAIN\_ARENA：记录该chunk是否属于主线程，1为不属于。
2. IS\_MAPPED：记录该chunk是否是由mmap分配的，1位是。
3. PREV\_INUSE：记录前一个chunk是否已分配，为1标识已分配。同时第一个chunk的PREV\_INUSE一定为1。当该标志位为0，则表明prev\_size变量里存的是前一个chunk的大小，否则就标识被征用了。

**注意，如果该chunk被分配了，那么下一个chunk的prev\_size是作为该chunk数据段的一部分存在的，然而那个4字节（或8字节）是不算进size里的。**

fd：如果该chunk被分配了，则fd及以后的变量都用来存该chunk的数据，丧失其本来的含义。当该chunk没被分配时，会被添加到对应的空闲管理链表里，fd则指向下一个（不一定物理相邻）空闲的chunk。

bk：该chunk被分配了则其丧失意义。否则则代表上一个（不一定物理相邻）空闲chunk。

fd\_nextsize：该chunk被分配了则其丧失意义。空闲时其用于较大的chunk，指向前一个（与fd不同）与当前chunk大小不同的第一个空闲块，不包含bin头指针。当chunk太小时，肯定用不到这个，例如说64位最小chunk size为32，则此时fd\_nextsize在该chunk中哪怕是空闲时也不存在。bk\_nextsize同理。

bk\_nextsize：该chunk被分配了则其丧失意义。空闲时其用于较大的chunk，指向后一个（与bk不同）与当前chunk大小不同的第一个空闲块，不包含bin头指针。

一般空闲的 large chunk 在 fd 的遍历顺序中，按照由大到小的顺序排列，即bk\_nextsize会越来越小。利用bk\_nextsize可以避免在寻找合适 chunk 时挨个遍历（直接跳到下一个大小不同的chunk）。

**5.4.4 bin**

主要见博客：<https://zhuanlan.zhihu.com/p/384220764>

分配或释放细节相关见：

https://github.com/jmpews/pwn2exploit/blob/master/PWN%E4%B9%8B%E5%A0%86%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86.md

关于last remainder参考下述链接中的实验

https://blog.csdn.net/qq\_41453285/article/details/97803141

关于源代码解读见https://zhuanlan.zhihu.com/p/390507862

bin主要分为了fast bin，small bin，large bin，unsorted bin。

虽然不在bin里面，但实际上还有top chunk和last remainder需要注意。

**5.4.5 unlink**

最终在博客https://zhuanlan.zhihu.com/p/63190549处看懂。

利用unlink必须让chunk满足以下要求（以下都以64位为基础）



设指向chunk的指针为ptr，因此可以得\*ptr = prev\_size。

由于FD=ptr->fd, BK=ptr->bk。故上述等式等价于

ptr->fd->bk=ptr; ptr->bk->fd=ptr;

即\*(ptr->fd+0x18)=ptr，\*(ptr->bk+0x10)=ptr

因此\*(FD+0x18)=ptr, \*(BK+0x10)=ptr

则FD+0x18=&ptr，BK+0x10=&ptr

要注意的是，上面两个式子的&ptr理论上是可以不相等的，因为没有声明只有一个内存地址里存的有ptr。例如说假如内存地址0x100里可以存ptr，内存地址0x200里也可以存ptr。但这里为了利用，我们不妨设两个式子的&ptr都等于0x100。

因此可得FD=&ptr-0x18; BK=&ptr-0x10

当度过了检查，下一步就会进入unlink的操作：

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

按照顺序来，当前内存地址100里存着ptr，即堆首地址。BK等于&ptr-0x10，而FD->bk=ptr，&(FD->bk)=0x100，即FD->bk=BK就是往内存地址0x100里写入值&ptr-0x10。

随后BK->fd=FD，由于&(BK->fd)=0x100，所以其含义就是往内存地址0x100处写入值&ptr-0x18。

因此在执行完成后，ptr的值变成了&ptr-0x18。先前ptr等于堆的首地址，现在ptr等于变量ptr本身的地址减0x18的值。即ptr=&ptr-0x18=0x100-0x18=0xE8

**5.4.6 Tcache**

**5.4.6.1 概览**

Tcache可以视为一个类似于fastbin的单向链表，只是他的check没有fastbin那么复杂，所以可以搞很多操作。 bin的数量是64个，64位系统以0x10递增，从24到1032字节。缓存的是非large chunk的chunk。Tcache的取出是最优先的。

Tcache里有个重要的数据结构tcache\_perthread\_struct，里面有两个成员，counts和entries。counts[n]记录了entries[n]链表的长度，或者说有多少个chunk。而entries[n]则直接指向了其下每个chunk的fd，即data段。

Tcache中有两个重要的函数tcache\_put和tcache\_get。

tcache\_put在请求分配大小不大于0x408且给定大小的tcache bin没满时使用，可以把一个chunk放到对应的tcache bin中。一个tcache bin最多可以放7个chunk。

tcache\_get仅检查tc\_idx是否越界，只要tc\_idx不大于等于64即可。

详细说明见https://blog.csdn.net/qq\_40890756/article/details/102560506

**5.4.6.2 使用**

free函数在检查释放块是否页对齐及前后堆块释放情况后，就优先把chunk放入tcache结构中。

内存分配的malloc函数中，有多种情况会把chunk往tcache里放：

1. 申请的内存块符合fastbin大小，且在fastbin内找到了可用的空闲块时，会把该fastbin链上其他内存块放到tcache里。要注意的是，由于从fastbin中取是从头取（LIFO），而取出之后插入到tcache中也是从头插入，因此从fastbin的chunk链转移到tcache后，chunk顺序会正好相反。
2. 申请内存块符合small bin大小，且在small bin中找到了可用的空闲块，会把该small bin链上其他空闲块放到tcache里。当放上去后，会对每个chunk的prev\_chunk置1防止被合并。同时由于small bin的获取是（FIFO）的，故会从尾部摘chunk，因此从small bin转移到tcache后chunk链顺序不变。
3. 当在unsorted bin链上循环处理时，找到大小合适的链时，不直接返回，而是先放到tcache里，继续处理。

Tcache的取出情况：在内存申请的开始部分，首先判断申请大小块在tcache中是否存在，存在就直接从tcache里取，否则再使用\_int\_malloc分配。

当malloc进入到unsorted bin分配时，按照上面③所说的，就算找到了合适的chunk也是先放到tcache里，然后继续处理后面的。

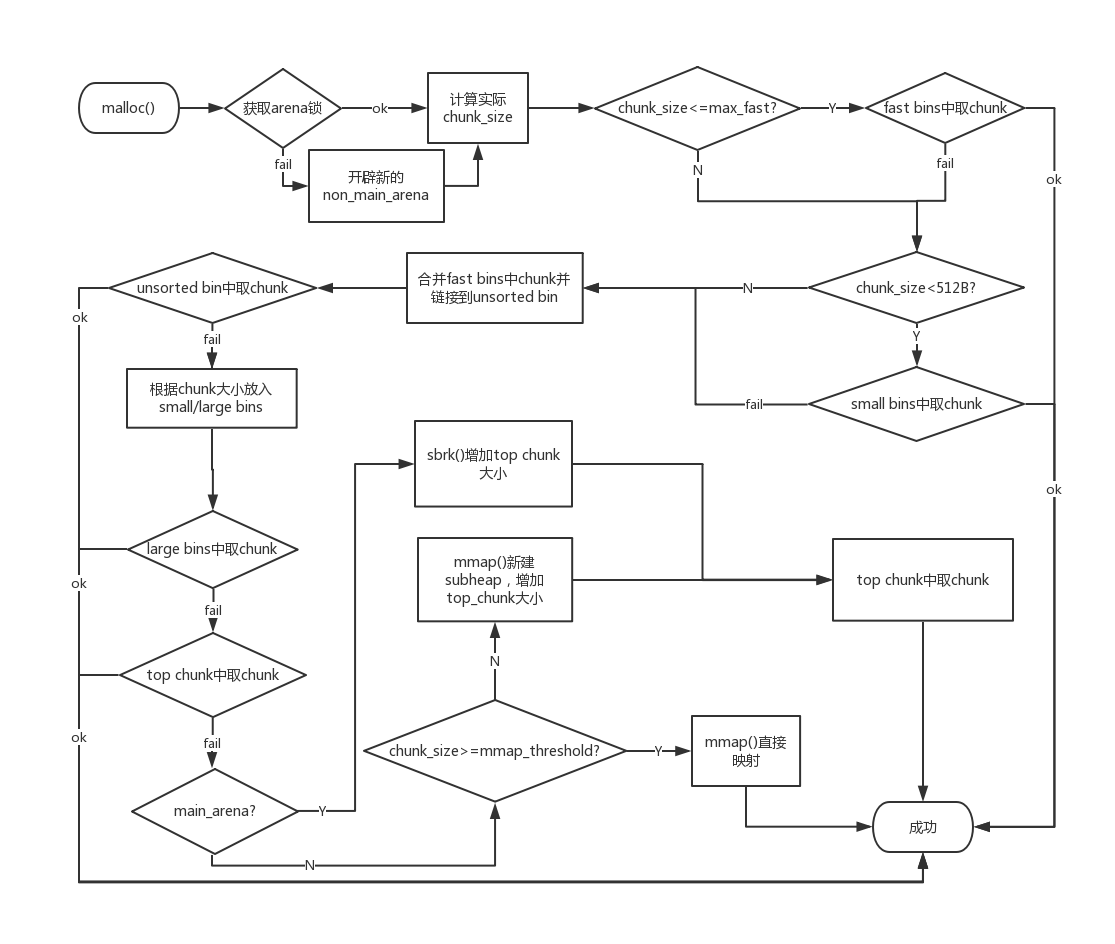
此时有如下情况



这段代码没有启动，因为limit为0。我的理解是，这里如果limit启用了，那么就是给count设置了一个上限阈值，当高于limit时就从tcache里拽出一个chunk甩出去。不过至少现在是没有用到这个代码。

当整个unsorted bin遍历完成后，如果之前找到了可以返回的chunk，那么这个chunk肯定已经被塞到tcache里了（暂时不清楚如果tcache满了怎么办，但哪怕满了，那个bin里也肯定都是能用的chunk），所以直接调用tcache\_get返回给用户使用。

**5.4.7 申请和释放流程**



流程图来源于网络，但都存在着一点问题，不过大的方向是对的，但有细节有问题。所以我简单概括一下。

堆分配流程看着比较复杂，简单概括一下重点就是以下几个步骤：

① chunk size <= max\_fast，从fast bin中取

② 大于max\_fast或fast bin中没能用的，从small bin中取

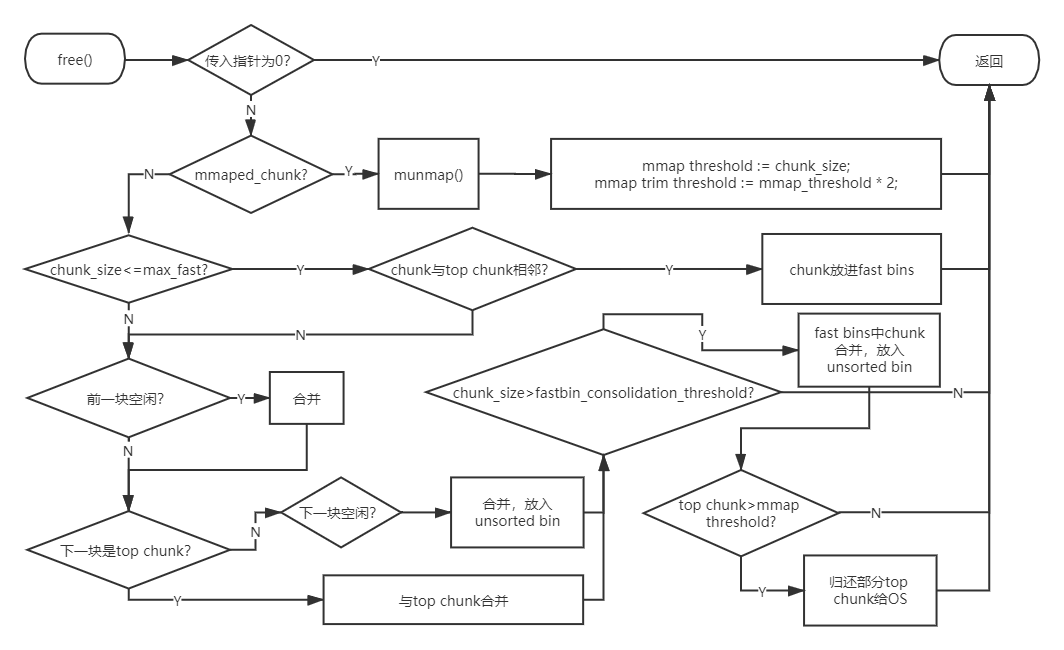
③如果 small bin中取不到，则直接进下一步。但如果size大于small bin的大小或者small bin还没初始化，会先合并fast bin中的chunk并放到unsorted bin中，然后从unsorted bin中取。

1. 在循环遍历unsorted bin的过程中，如果找到size满足的，就直接返回。如果没找到，就会顺便把这个chunk放到他该有的位置（small/large bin）中。但由于找到size满足就返回了，所以不一定会把所有的chunk都分配出去。要注意的是，如果只有last remainder且发现可以切割，就会切割。剩余部分放在unsorted bin里。
2. 如果申请的大小小于large bin size，即small bin size的大小。那此时说明对应的small bin中一定没有这个chunk，因为如果有，要么在第二步要么在第四步就拿到了。因此只能从比当前要求的size大的bin中找。此时如果找到了，那就肯定可以切割，切割出来一部分送出去，一部分当last remainder送进unsorted bin

⑤ 从large bin中取chunk，如果小于预期chunk，就切割。

⑥ 取不到就从top chunk中分割chunk

⑦ 还取不到就用mmap向系统申请，这样便于free该块的时候可以直接还给系统。



堆回收的流程大概如下，流程图部分地方有问题，以叙述为准：

① 传入指针为空就返回

② 如果是mmaped\_chunk，就走mumap流程回收

③ 如果小于等于max\_fast，就直接把chunk塞进fast bin里，这里图上有点问题

④ 如果大于max\_fast，就走合并流程，先合并前一块空闲块，然后合并后一块空闲块。如果后一块是top chunk，则直接并入top chunk。如果不是就合并后放到unsorted bin中

⑤ 如果当前这个chunk size太大，超过了阈值。则合并fast bin中所有的chunk并放到unsorted bin中。

**5.5 关于WORD，DWORD，QWORD等**

对于CPU而言，“字”一词有特殊含义，字即WORD。无论寄存器多长（无论多少位系统），“字”长都是2字节，即WORD=2B，DWORD=4B，QWORD=8B。

db：define Byte，1字节

dw：define word，2字节

dd：define double（word），4字节

**5.6 关于alsr和pie的差异**

