动态调试

gcc编译

参数-O：优化，比如一些不可能(默认无栈溢出等情况)执行的代码直接忽略，且指定输出文件的名称

参数-Wall：编译器在编译过程中输出警告信息

参数-g：编译器输出调试(debug)信息

```
gcc -m32 -o test test.c
gcc test.c -o test
gcc -Wall -g -o test test.c #-wall生成警告信息
gcc -no-pie -fno-stack-protector -z execstack -m32 -o read read.c #关闭地址随机化，关闭栈保护
gcc -no-pie -fstack-protector-all printf1.c -o printf1 #-all，全部函数都加canary保护
```

objdump：用来查看目标文件或者可执行的目标文件的构成的gcc工具

```
objdump -t -j .text read #查看read程序的.text段有哪些函数
```

gdb动态调试

(19条消息) linux下gdb调试方法与技巧整理*花开蝶自来-liu的博客-CSDN博客*gdb调试

GDB常用命令大全 GDB 命令详细解释

http://c.biancheng.net/gdb/

```
b                    #打断点，具体地址前要加*
n                    #下一步
i b                  #查看断点
i r                  #查看寄存器
r                    #运行
d 断点编号           #删除断点
enable b 断点编号    #取消断点
stack 0x20           #查看栈
disassemble $rip     #rip所在的函数整个编译出来
set ($rbp-0x10)=0x61
finish               #执行完当前函数，返回调用，跳出函数
```

x和p类似

```
x/xw addr    显示某个地址处开始的16进制内容，如果有符号表会加载符号表
x/x $esp     查看esp寄存器中的值
x/s addr     查看addr处的字符串
x/b addr     查看addr处的字符
x/i addr     查看addr处的反汇编结果
x/20i $rip   i是汇编，从rip开始编译20行。b是byte
p $rbp               #打印值，十进制，若要十六进制输出p改为p/x
```

disassemble：指定要反汇编的函数。如果指定，反汇编命令将产生整个函数的反汇编输出。可简写为disass

and：堆栈内存对齐，程序访问的地址必须向16字节对齐（被16整除），内存对齐后，可以提高访问效率。格式and esp,0xFFFFFFF0

vmmap：查看段的权限

不调试直接查看

```
odjdump -t xxx # 查看程序中使用的函数
odjdump -d xxx #查看程序中函数的汇编代码
odjdump -d -j .plt xxx #查看plt表
# -j的参数有 .text代码段、.const只读数据段、.data读写数据段、.bss bss段
objdump -d -M intel xxx #查看程序中函数的汇编代码，并且汇编代码是intel的
```

查看段

```
readelf 文件名
```

ida远程调试

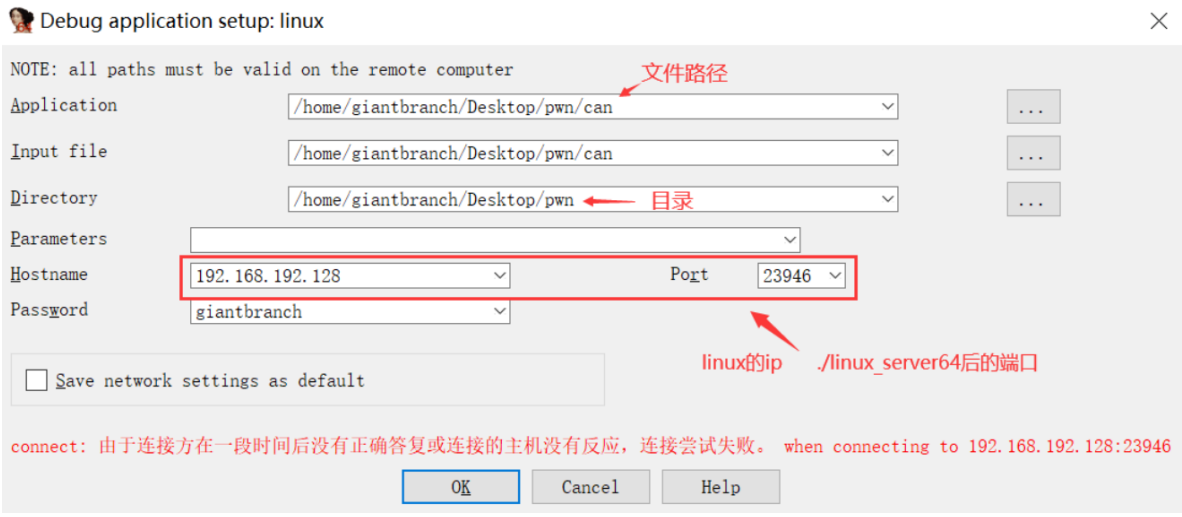
> https://ghidra-sre.org/ ubuntu下反编译工具

把ida中的linux_server或者linux_server64移到linux的程序目录下

linux下

```
./linux_server64
```

然后ida里选择remote linux debugger



然后就可以开始调试了，输出是在linux里

汇编

寄存器

32位寄存器

PWN常用寄存器——esp，edp，eip

```
esp：栈顶地址，在压栈和退栈时发生变化
ebp：当前函数状态的基地址，在函数运行时不变，可以用来索引确定函数参数或局部变量的位置
eip：用来存储即将执行的程序指令的地址
rax：通用寄存器，存放函数返回值
```

用checksec检查程序

```
checksec 可执行文件名
```

寄存器标志位

16位

8086CPU 的 flag 寄存器的结构如图 11.1 所示。

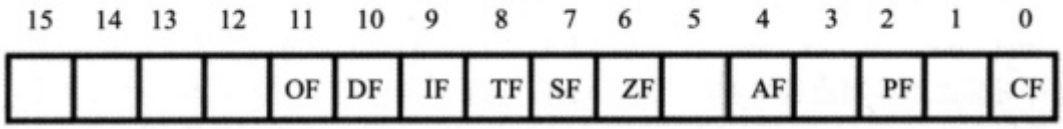
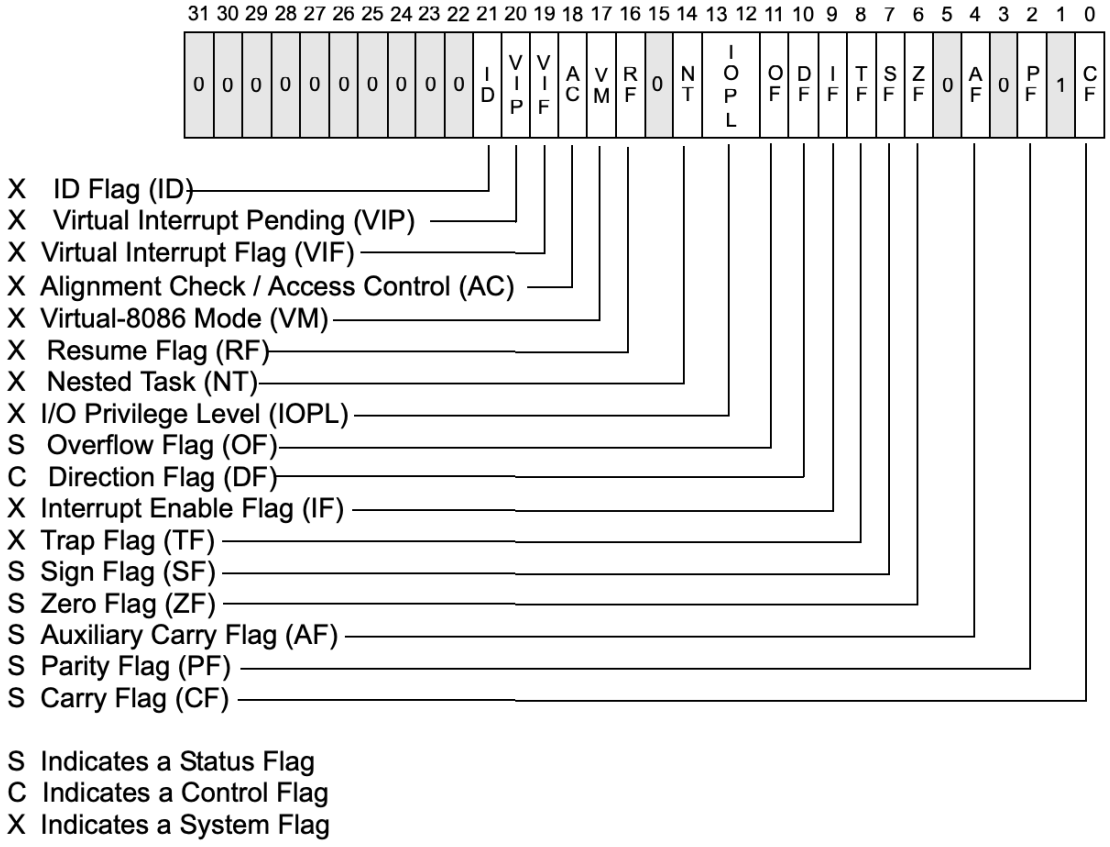


图 11.1　flag 寄存器各位示意图

flag 的 1、3、5、12、13、14、15 位在 8086CPU 中没有使用，不具有任何含义。0、2、4、6、7、8、9、10、11 位都具有特殊的含义。

32位

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VM	RF	--	NT	IOPL	IOPL	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF



汇编语言

32位x86架构下汇编指令有两种格式：intel和AT&T

- Intel：寄存器名称和数值前无符号
- AT&T：寄存器名称前加“%”，数值前加“$”

指令	具体操作	格式
mov	数据传输指令，将src传至dst	mov dst,src
pop	弹出堆栈指令，将栈顶的数据弹出并存至dst	pop dst
add/sub	加减法指令，将运算结果存到dst	add/sub dst,src
push	压入栈堆指令，将src压入栈内	push src
lea	取地址指令，将mem的地址存至reg	lea reg,mem
call	调用指令，将当前的eip压入栈顶，并将ptr存入eip	call ptr
leave	等于mov+pop	

```
mov    rbp,rsp          #rbp=rsp cmp
movq   %rsp,%rbp
lea    rax,[rbp-0x18]   #rax=rbp-0x18
xor    ebx,ebx          #异或，ebx=0
cmp判断相等，下面一般跟JCC系列
add
```

JCC系列，跟标记位有关系，跳转用的，现搜就行

常见的保护

CANARY(栈保护)

栈溢出保护是一种缓冲区溢出攻击缓解手段，当函数存在缓冲区溢出攻击漏洞时，攻击者可以覆盖栈上的返回地址来让shellcode能够得到执行。当启用栈保护后，函数开始执行的时候会先往栈里插入cookie信息，当函数返回的时候会验证cookie信息是否合法，如果不合法就停止程序运行。攻击者在覆盖返回地址的时候往往也会将cookie信息给覆盖掉，导致栈保护检查失败而组织shellcode的执行。在linux中我们将cookie信息称为canary

gcc在4.2版本中添加了-fstack-protector和-fstack-protector-all编译参数以支持栈保护功能，4.9新增了-fstack-protector-strong编译参数让保护的范围更广。

```
gcc -fno-stack-protector -o test test.c  //禁用栈保护
gcc -fstack-protector -o test test.c  //启用堆栈保护，不过只为局部变量中含有 char 数组的函数插入保护代码
gcc -fstack-protector-all -o test test.c //启用堆栈保护，为所有函数插入保护代码
```

FORTIFY

fortify和栈保护都是gcc的为了增强保护的一种机制，防止缓冲区溢出攻击。并不常见

NX(DEP)

NX即No-execute(不可执行)，NX基本原理是将数据所在的内存页表示为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行saz指令，此时CPU就会抛出异常而不是执行恶意指令

```
gcc -z execstack -o test test.c //关闭NX保护
```

PIE(ASLR)

一般情况下NX(Windows平台上称其为DEP)和地址空间的分布随机化(ASLR)会同时工作。
内存地址随机化机制有以下三种情况：

- 0-表示关闭进程地址空间随机化
- 1-表示将mmap的基址，stack和vdso页面随机化
- 2-表示在1的基础上增加栈(heap)的随机化

可以方法基于Ret2libc方式的针对DEP的攻击。ASLR和DEP配合使用，能有效阻止攻击者在堆栈上运行恶意代码

RELRO

设置符号重定向表格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）攻击。RELRO为” Partial RELRO”，说明我们对GOT表具有写权限。

canary保护

有canary保护下，在ebp上面会再压入一个canry的值

PIE与ASLR

关于Linux下ASLR与PIE的一些理解 - rec0rd - 博客园 (cnblogs.com)

ASLR和PIE的区别和作用*coke_pwn的博客-CSDN博客*aslr

首先，ASLR的是操作系统的功能选项，作用于executable（ELF）装入内存运行时，因而只能随机化stack、heap、libraries（堆栈和libc）的基址；而PIE（Position Independent Executables）是编译器（gcc，..）功能选项（-fPIE），作用于excutable编译过程，可将其理解为特殊的PIC（so专用，Position Independent Code），加了PIE选项编译出来的ELF用file命令查看会显示其为so，其随机化了ELF装载内存的基址（代码段、plt、got、data等共同的基址）。

其次，ASLR早于PIE出现，所以有return-to-plt、got hijack、stack-pivot(bypass stack ransomize)等绕过ASLR的技术；而在ASLR+PIE之后，这些bypass技术就都失效了，只能借助其他的信息泄露漏洞泄露基址（常用libc基址）。

最后，ASLR有0/1/2三种级别，其中0表示ASLR未开启，1表示随机化stack、libraries，2还会随机化heap。

ASLR 不负责代码段以及数据段的随机化工作，这项工作由 PIE 负责。但是只有在开启 ASLR 之后，PIE 才会生效。

	作用位置	归属	作用时间
ASLR	1：栈基地址（stack）、共享库（.so\libraries）、mmap 基地址 2：在1基础上，增加随机化堆基地址（chunk）	系统功能	作用于程序（ELF）装入内存运行时
PIE	代码段（ .text ）、初始化数据段（ .data ）、未初始化数据段（ .bss ）	编译器功能	作用于程序（ELF）编译过程中

在传统的操作系统里，用户程序的地址空间布局是固定的，自低向高依次为代码区， BSS区，堆栈区，攻击者通过分析能轻易得出各区域的基地址，在此情况下，只要攻击者的 注入代码被执行，攻击者就能随意跳转到想到达的区域，终取得计算机的控制权，试想如 果一个程序在执行时，系统分配给此进程三个区域的基地址是随机产生的，每次该程序执行 都不一样，那么攻击者注入的代码即使被执行，也终会

因为无法找到合法的返回地址而产 生错误，终进程停止，攻击者无法入侵，这就是地址空间布局随机化的基本思想。

栈溢出

超详细]栈溢出漏洞原理实例讲解*breezeO_o的博客-CSDN博客*栈溢出漏洞原理

https://www.cnblogs.com/ichunqiu/p/15476743.html

原理：栈从高地址向低地址生长，但局部变量从低地址向高地址写。未控制好局部变量的值会导致，就可以覆盖返回地址

缓冲区溢出

编写程序时没有考虑到控制或者错误控制用户输入的长度，本质就是向定长的缓冲区中写入了超长的数据，造成超出的数据覆写了合法内存区域

- 栈溢出——最常见、漏洞比例最高、危害最大的二进制漏洞
- 堆溢出——堆管理器复杂，利用样式繁多
- Data段溢出（如bss段）——攻击效果依赖于Data段上存放了何种控制数据

攻击流程

```
checksec text       #先将文件放到虚拟机里查看保护措施
sudo chmod +x text  #修改可执行权限
```

IDA查看

通常用gets()读取用户输入，容易发生栈溢出

写脚本

```
from pwn import *
p=process("./text")  #本地连接
p=remote("ip",port)  #远程连接
backdoor=0x400677    #后门函数的地址
payload='a'*0x18+p64(backdoor)
p.recvuntil("Your seggestion:\n") #原函数gets上有一行puts("Your seggestion"),puts会加换行所以这里\n
# 已经建立进程交互，不断读取进程输出直到引号内容recvuntil会让程序停止
p.sendline(payload)  #发送一行攻击数据
p.interactive()      #通过终端交互而不是脚本继续交互
```

运行脚本

```
python 1.py
```

ret2xx

rop

ROP系统攻击_百度百科 (baidu.com)

> ROP全称为Return-oriented Programming（面向返回的编程）是一种新型的基于代码复用技术的攻击，攻击者从已有的库或可执行文件中提取指令片段，构建恶意代码。

rop主要思想在栈缓冲区溢出的基础上，利用程序中已有的小片段(gadgets)来改变某些寄存器或者变量的值，从而控制程序的执行流程。所谓gadgets就是以ret结尾的指令序列，通过这些指令序列，我们可以修改某些地址的内容，方便控制程序的执行流程。

rop核心在于利用了指令集中的ret指令，改变了指令流的执行顺序。

需要满足条件：

- 程序存在溢出，且可以控制返回地址
- 可以找到满足条件的gadgets以及相应的gadgets地址（如果gadgets每次地址是不固定的，则需要动态获取相应地址）

ret2text

控制程序执行本身已有的代码(.text)

栈溢出触发

```
gets(buf);
strcpy(dest,src);
scanf("%s",buf);
strcat(buf,buf2);
read(0,buf,size);
```

快速定位偏移

pwndbg快速定位

```
cyclic 200 #生产大量数据
cyclic -l 0x6161616c #查询报错的地址
```

peda快速定位

```
pattern create 200
pattern offset 0x41414641
```

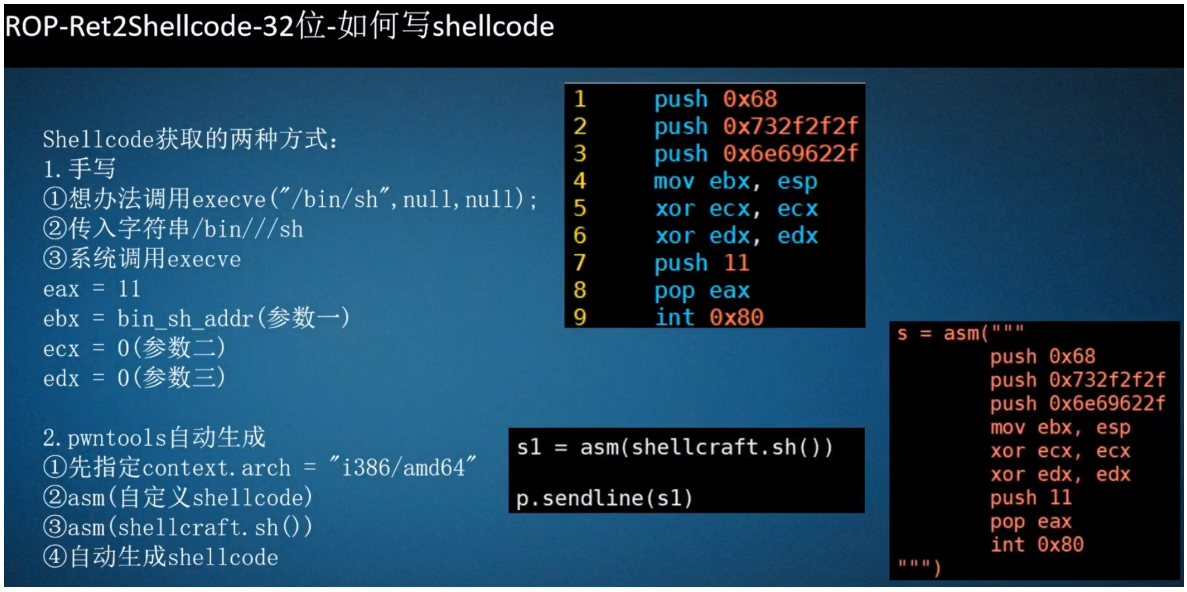
ret2shellcode

控制程序执行shellcode代码，shellcode指用于完成某个功能的汇编代码，常见的功能主要是获取目标系统的shell

linux系统调用号

linux下/usr/incude/x86_64-linux-gnu/asm/unistd_64.h(unistd_32.h)

获取shellcode



三个反斜杠，因为位数不够，会自动填充00，防止00截断

shellcode模板

执行到int 0x80后会执行eax

32位

eax	系统调用号
ebx	第1个参数
ecx	2
edx	3
esi	4
edi	第5个参数
int 0x80	

64位

rax	系统调用号
rdi	第1个参数
rsi	2
rdx	3
rcx	4
r8	5
r9	第6个参数
syscall	

ret2syscall

控制程序执行系统调用

gets函数存在栈溢出，且没有长度限制，直到读到0x0A(回车)

思路：一般静态编译程序会存在int 0x80，可以使用中断调用系统函数

- read(1,bss+0x100,8)读入/bin/sh
- execve(bss+0x100,0,0) getshell

ROPgadget

从程序中寻找可以利用的小片段，拼接成shellcode

ret2Libc

system函数属于libc，而libc.so动态链接库中的函数之间相对偏移是固定。即使有ASLR保护，也只是针对地址中间位进行随机，最低的12位并不会发生改变

使用条件

- 可以泄露libc函数地址： 程序有输出函数如printf、puts、write
- 设置好参数为某函数GOT表地址（GOT表中保存已调用过的函数的真实地址）

先泄露任意一个函数的真实地址，只有被执行过的函数才能获取地址。获取libc版本，用libc database search (blukat.me)或LibcSearcherlieanu/LibcSearcher: glibc offset search for ctf. (github.com)。根据偏移量获取shell和sh位置，用libc基地址（函数动态地址-函数偏移量）计算其他函数的地址（基地址+函数偏移量）。最终执行程序获取shell

libc

libc是Linux下的C函数库，libc中包含着各种常用的函数，在程序执行的时候才被加载到内存中。libc一定可执行，跳转到libc中函数绕过NX保护

libc函数的位置

- ASLR地址随机化：系统开启 /proc/sys/kernel/randomize_va_space
- 0表示关闭ASLR
- 1表示保留的随机化，共享库、栈、mmap0以及VDSO随机化
- 2表示完全的随机化，在1的基础上通过brk()分配的内存和空间也将被随机化

Linux延迟绑定机制

程序的动态链接（3）：延迟绑定*Aspiresky的博客-CSDN博客*动态链接 延迟绑定

GOT表和PLT表知识详解*77458的博客-CSDN博客*got表

plt表：程序联动表（内部函数表）

got表：全局偏移表（全局函数表）

动态链接的程序调用了libc的库函数，但是libc在运行的时候才被加载到内存中调用libc函数时，才解析出函数在内存中的地址

延迟绑定机制的实现

- 所有函数调用的libc函数都有对应的PLT表和GOT表，其位置固定
- PLT表：
    - 调用函数call puts@plt
    - PLT表中存放着指令：jmp [puts_got]
- GOT表：解析函数真实地址放入GOT表中存储

GOT表中包含libc函数真实地址，用于泄露地址。覆盖新地址到GOT表，劫持函数流程

不用知道libc函数真实地址，使用PLT地址即可调用函数

杂项

切换

pwndbg和pwn-peda的切换

去~/peda/目录 找peda.py改名就行
改成ped.py就可以用pwndbg

setvbuf函数做优化用的，不用管

大小端

最左端-最高有效字节(MSB)、最右端-最低有效字节(LSB)

大端

- 低地址(MSB)——高地址(LSB)

小端

- 低地址(LSB)——高地址(MSB)

CPU每次只处理8位，从最低有效字节开始处理

调用约定

32位是直接从栈上依次

Linux 下的 x64 调用约定

> 一个函数在调用时，如果参数个数小于等于 6 个时，前 6 个参数是从左至右依次存放于 RDI，RSI，RDX，RCX，R8，R9 寄存器里面，剩下的参数通过栈传递，从右至左顺序入栈；

shellcode

shellcode是软件漏洞利用过程中使用的一小段机器代码

下面是一段C程序，执行后可以使用shell

```
//gcc test2.c -o test2
#include "stdlib.h"
#include "unistd.h"
void main(){
    system("/bin/sh");
    exit(0);
}
```