

什么是 pwn

破解、利用成功（程序的二进制漏洞）

攻破（设备、服务器）

控制（设备、服务器）

CTF 比赛主要表现以下几个技能上：逆向工程、密码学、Web 漏洞、二进制溢出、网络和取证等。在国际 CTF 赛事中，二进制溢出也称之为 PWN。

PWN 是一个黑客语法的俚语词，自"own"这个字引申出来的，这个词的含意在于，玩家在整个游戏对战中处在胜利的优势，或是说明竞争对手处在完全惨败的情形下，这个词习惯上在网络游戏文化主要用于嘲笑竞争对手在整个游戏对战中已经完全被击败（例如："You just got pwned!"）。有一个非常著名的国际赛事叫做 Pwn2Own，相信你现在已经能够理解这个名字的含义了，即通过打败对手来达到拥有的目的。

CTF 中 PWN 题型通常会直接给定一个已经编译好的二进制程序（Windows 下的 EXE 或者 Linux 下的 ELF 文件等），然后参赛选手通过对二进制程序进行逆向分析和调试来找到利用漏洞，并编写利用代码，通过远程代码执行来达到溢出攻击的效果，最终拿到目标机器的 shell 夺取 flag。

exploit

用于攻击的脚本与方案

payload

攻击载荷，是的目标进程被劫持控制流的数据

shellcode

调用攻击目标的 shell 的代码

前置知识

bss 段：

bss 段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。

bss 是英文 Block Started by Symbol 的简称。

bss 段属于静态内存分配。

data 段：

数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。

数据段属于静态内存分配。

text 段：

代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。

这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读（某些架构也允许代码段为可写，即允许修改程序）。

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆（heap）：

堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。

当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上

（堆被扩张）；

当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。

栈(stack)：

栈又称堆栈，是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在数据段(.data)中存放变量）。

除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。

由于栈的先进先出(FIFO)特点，所以栈特别方便用来保存/恢复调用现场。

从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。一个程序本质上都是由 bss 段、data 段、text 段三个组成的。

这样的概念，不知道最初来源于哪里规定，但在当前的计算机程序设计中是很重要的一个基本概念。

而且在嵌入式系统的设计中也非常重要，牵涉到嵌入式系统运行时的内存大小分配，存储单元占用空间大小的问题。

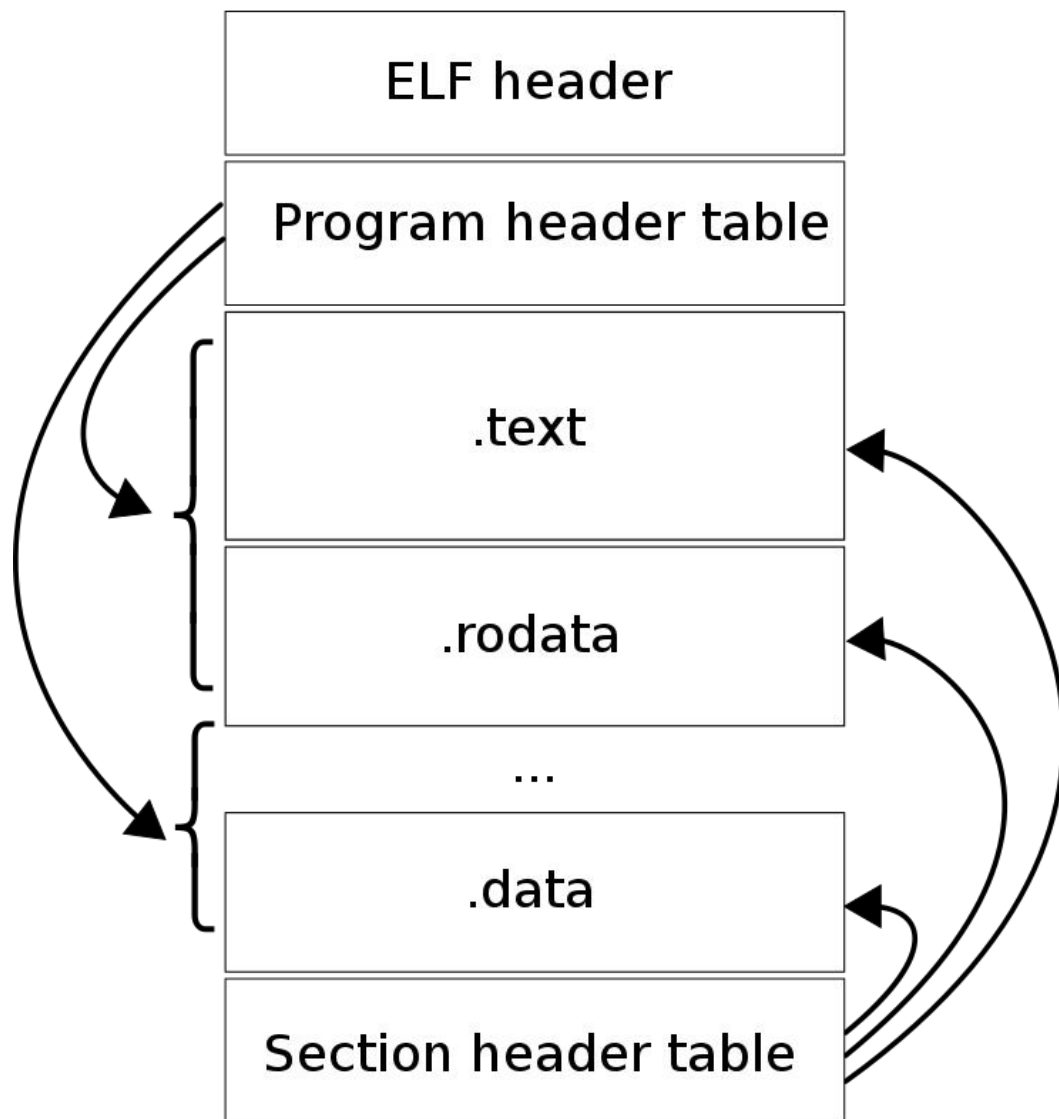
在采用段式内存管理的架构中（比如 intel 的 80x86 系统），bss 段通常是指用来存放程序中未初始化的全局变量的一块内存区域，

一般在初始化时 bss 段部分将会清零。bss 段属于静态内存分配，即程序一开始就将其清零了。

比如，在 C 语言之类的程序编译完成之后，已初始化的全局变量保存在.data 段中，未初始化的全局变量保存在.bss 段中。

text 和 data 段都在可执行文件中（在嵌入式系统里一般是固化在镜像文件中），由系统从可执行文件中加载；

而 bss 段不在可执行文件中，由系统初始化。



ELF 文件头表（ELF header）

记录了 ELF 文件的组织结构

程序头表/段表（Program header table）

告诉系统如何创建进程

生成进程的可执行文件必须拥有此结构

重定位文件不一定需要

节头表（Section header table）

记录了 ELF 文件的节区信息

用于链接的目标文件必须拥有此结构

其它类型目标文件不一定需要

程序 1

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

```
int ar[30000];
```

```
int main(){
```

```
    return 0;
```

```
}
```

程序 2


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

```
int ar[300000] = {1, 2, 3, 4, 5, 6 };
```

```
int main(){
```

```
    return 0;
```

```
}
```

A terminal window with a dark blue background and white text. It shows three lines of commands: 'pwn> gcc 1.cpp -o 1.exe', 'pwn> gcc ./2.cpp -o 2.exe', and 'pwn>'.

```
pwn> gcc 1.cpp -o 1.exe
pwn> gcc ./2.cpp -o 2.exe
pwn>
```

StudyPE+ (x64) 1.10 beta 2 --> 1.exe

文件 选项 工具 杂项 帮助

[概况] [PE头] [数据表] [区段] [导入] [导出] [资源] [重定位] [异常] [.Net]

名称	Virtual Address	Virtual Size	Raw Address	Raw Size	Characteristics
.text	00001000	00002B94	00000400	00002C00	60500060
.data	00004000	0000001C	00003000	00000200	C0300040
.rdata	00005000	000002D0	00003200	00000400	40300040
/4	00006000	000009A4	00003600	00000A00	40300040
.bss	00007000	0001D530	00000000	00000000	C0600080
.idata	00025000	000005A8	00004000	00000600	C0300040
.CRT	00026000	00000018	00004600	00000200	C0300040
.tls	00027000	00000020	00004800	00000200	C0300040
/14	00028000	00000038	00004A00	00000200	42400040
/29	00029000	00001CFF	00004C00	00001E00	42100040
/41	0002B000	0000012F	00006A00	00000200	42100040
/55	0002C000	000001C8	00006C00	00000200	42100040
/67	0002D000	00000038	00006E00	00000200	42300040

☐ 显示附加数据或数字

查看数据表 查看区段 删除区段 编辑

13:53:53

StudyPE+ (x64) 1.10 beta 2 --> 2.exe

文件 选项 工具 杂项 帮助

[概况] [PE头] [数据表] [区段] [导入] [导出] [资源] [重定位] [异常] [.Net]

名称	Virtual Address	Virtual Size	Raw Address	Raw Size	Characteristics
.text	00001000	00002B94	00000400	00002C00	60500060
.data	00004000	00124FB8	00003000	00125000	C0600040
.rdata	00129000	000002D0	00128000	00000400	40300040
/4	0012A000	000009A4	00128400	00000A00	40300040
.bss	0012B000	00000070	00000000	00000000	C0300080
.idata	0012C000	000005A8	00128E00	00000600	C0300040
.CRT	0012D000	00000018	00129400	00000200	C0300040
.tls	0012E000	00000020	00129600	00000200	C0300040
/14	0012F000	00000038	00129800	00000200	42400040
/29	00130000	00001CFF	00129A00	00001E00	42100040
/41	00132000	0000012F	0012B800	00000200	42100040
/55	00133000	000001C8	0012BA00	00000200	42100040
/67	00134000	00000038	0012BC00	00000200	42300040

☐ 显示附加数据或数字

查看数据表 查看区段 删除区段 编辑

13:54:17

可以看到两者的区别在于 data 段的大小：  
 全局的未初始化变量存在于 .bss 段中，具体体现为一个占位符；（第一个程序）  
 全局的已初始化变量存于 .data 段中；（第二个程序）

.bss 是不占用 .exe 文件空间的，其内容由操作系统初始化（清零）；  
.data 却需要占用，其内容由程序初始化。因此造成了上述情况。  
bss 段（未手动初始化的数据）并不给该段的数据分配空间，只是记录数据所需空间的大小；  
bss 段的大小从可执行文件中得到，然后链接器得到这个大小的内存块，紧跟在数据段后面。  
data 段（已手动初始化的数据）则为数据分配空间，数据保存在目标文件中；  
data 段包含经过初始化的全局变量以及它们的值。当这个内存区进入程序的地址空间后全部清零。  
包含 data 段和 bss 段的整个区段此时通常称为数据区。

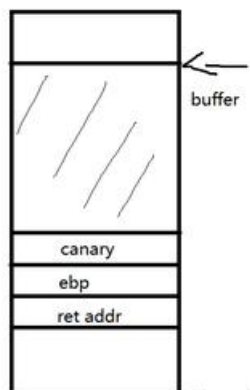
## PWN 文件保护机制

Linux ELF 文件的保护主要有四种：Canary、NX、PIE、RELRO

### 一：Canary

Canary 是金丝雀的意思。技术上表示最先的测试的意思。这个来自以前挖煤的时候，矿工都会先把金丝雀放进矿洞，或者挖煤的时候一直带着金丝雀。金丝雀对甲烷和一氧化碳浓度比较敏感，会先报警。所以大家都用 Canary 来搞最先的测试。Stack Canary 表示栈的报警保护。

在函数返回值之前添加的一串随机数（不超过机器字长）（也叫做 cookie），末位为 /x00（提供了覆盖最后一字节输出泄露 Canary 的可能），如果出现缓冲区溢出攻击，覆盖内容覆盖到 Canary 处，就会改变原本该处的数值，当程序执行到此处时，会检查 Canary 值是否跟开始的值一样，如果不一样，程序会崩溃，从而达到保护返回地址的目的。



总的来说，Canary 参数表示着对栈的保护，防止栈溢出的一种保护，即在栈靠近栈底某个位置设置初值，防止栈溢出的一种保护

GCC 用法：

`gcc -o test test.c // 默认情况下，不开启 Canary 保护`

`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`

`-fno-stack-protector /-fstack-protector / -fstack-protector-all (关闭 / 开启 / 全开启)`

## 二：NX

NX 即 No-eXecute（不可执行）的意思，NX（DEP）的基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入 shellcode 时，程序会尝试在数据页面上执行指令，此时 CPU 就会抛出异常，而不是去执行恶意指令。

正常在栈溢出时通过跳转指令跳转至 shellcode，但是 NX 开启后 CPU 会对数据区域进行检查，当发现正常程序不执行，并跳转至其他地址后会抛出异常，接下来不会继续执行 shellcode，而是去转入异常处理，处理后会禁止 shellcode 继续执行

GCC 用法：

`gcc -o test test.c // 默认情况下，开启 NX 保护`

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

`gcc -z noexecstack -o test test.c // 开启 NX 保护`

`-z execstack / -z noexecstack (关闭 / 开启)`

## 三：PIE（ASLR）

一般情况下 NX（Windows 平台上称为 DEP）和地址空间分布随机化（PIE/ASLR）（address space layout randomization）会同时工作。内存地址随机化机制有三种情况：

0 - 表示关闭进程地址空间随机化。

1 - 表示将 mmap 的基地址，栈基地址和 .so 地址随机化

2 - 表示在 1 的基础上增加 heap 的地址随机化

该保护能使每次运行的程序的地址都不同，防止根据固定地址来写 exp 执行攻击。

可以防止 Ret2libc 方式针对 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

linux 下关闭 PIE 的命令如下：

`sudo -s echo 0 > /proc/sys/kernel/randomize_va_space`

GCC 用法：

`gcc -o test test.c // 默认情况下，不开启 PIE`

`gcc -fPIE -pie -o test test.c // 开启 PIE，此时强度为 1`

`gcc -fPIE -pie -o test test.c // 开启 PIE，此时为最高强度 2`

`gcc -fpic -o test test.c // 开启 PIC，此时强度为 1，不会开启 PIE`

`gcc -fPIC -o test test.c // 开启 PIC，此时为最高强度 2，不会开启 PIE`

`-no-pie / -pie (关闭 / 开启)`

## 四：RELRO

Relocation Read-Only (RELRO) 可以使程序某些部分成为只读的。它分为两种：Partial RELRO 和 Full RELRO，即：部分 RELRO 和 完全 RELRO。

部分 RELRO 是 GCC 的默认设置，几乎所有的二进制文件都至少使用部分 RELRO。这样仅仅只能防止全局变量上的缓冲区溢出从而覆盖 GOT。

完全 RELRO 使整个 GOT 只读，从而无法被覆盖，但这样会大大增加程序的启动时间，因为程序在启动之前需要解析所有的符号。

在 Linux 系统安全领域数据可以写的存储区就会是攻击的目标，尤其是存储函数指针的区域。所以在安全防护的角度应尽量减少可写的存储区域

RELRO 会设置符号重定向表格为只读或者程序启动时就解析并绑定所有动态符号，从而减少对 GOT 表的攻击。如果 RELRO 为 Partial RELRO，就说明对 GOT 表具有写权限

主要用来保护重定位表段对应数据区域，默认可写

Partial RELRO: .got 不可写，got.plt 可写

Full RELRO: .got 和 got.plt 不可写

got.plt 可以简称为 got 表

GCC 用法:

gcc -o test test.c // 默认情况下, 是 Partial RELRO

gcc -z norelro -o test test.c // 关闭, 即 No RELRO

gcc -z lazy -o test test.c // 部分开启, 即 Partial RELRO

gcc -z now -o test test.c // 全部开启

-z norelro / -z lazy / -z now (关闭 / 部分开启 / 完全开启)

GOT 表和 PLT 表将在下一节进行介绍

## 五、FORTIFY

fortify 是轻微的检查, 用于检查是否存在缓冲区溢出的错误。适用于程序采用大量的字符串或者内存操作函数, 如:

memcpy():

描述: void \*memcpy(void \*str1, const void \*str2, size\_t n)

从存储区 str2 复制 n 个字符到存储区 str1

参数: str1 -- 指向用于存储复制内容的目标数组, 类型强制转换为 void\* 指针

str2 -- 指向要复制的数据源, 类型强制转换为 void\* 指针

n -- 要被复制的字节数

返回值: 该函数返回一个指向目标存储区 str1 的指针

memset():

描述: void \*memset(void \*str, int c, size\_t n)

复制字符 c (一个无符号字符) 到参数 str 所指向的字符串的前 n 个字符

参数: str -- 指向要填充的内存块

c -- 要被设置的值。该值以 int 形式传递, 但是函数在填充内存块时是使用该值的无符号字符形式

n -- 要被设置为该值的字节数

返回值: 该值返回一个指向存储区 str 的指针

strcpy():

描述: char \*strcpy(char \*dest, const char \*src)

把 src 所指向的字符串复制到 dest, 容易出现溢出

参数: dest -- 指向用于存储复制内容的目标数组

src -- 要复制的字符串

返回值: 该函数返回一个指向最终的目标字符串 dest 的指针

strcpy():

描述: extern char \*strcpy(char \*dest, char \*src)

把 src 所指由 NULL 借宿的字符串复制到 dest 所指的数组中

说明: src 和 dest 所指内存区域不可以重叠且 dest 必须有足够的空间来容纳 src 的字符串  
返回指向 dest 结尾处字符 (NULL) 的指针

返回值:

strncpy():

描述: char \*strncpy(char \*dest, const char \*src, size\_t n)

把 src 所指向的字符串复制到 dest, 最多复制 n 个字符。当 src 的长度小于 n 时, dest 的剩余部分将用空字节填充

参数: dest -- 指向用于存储复制内容的目标数组

src -- 要复制的字符串



n -- 要从源中复制的字符数

返回值：该函数返回最终复制的字符串

**strcat():**

描述：char \*strcat(char \*dest, const char \*src)

把 src 所指向的字符串追加到 dest 所指向的字符串的结尾

参数：dest -- 指向目标数组，该数组包含了一个 C 字符串，且足够容纳追加后的字符串

src -- 指向要追加的字符串，该字符串不会覆盖目标字符串

返回值：

**strncat():**

描述：char \*strncat(char \*dest, const char \*src, size\_t n)

把 src 所指向的字符串追加到 dest 所指向的字符串的结尾，直到 n 字符长度为止

参数：dest -- 指向目标数组，该数组包含了一个 C 字符串，且足够容纳追加后的字符串，包括额外的空字符

src -- 要追加的字符串

n -- 要追加的最大字符数

返回值：该函数返回一个指向最终的目标字符串 dest 的指针

**sprintf():PHP**

描述：sprintf(format,arg1,arg2,arg++)

arg1、arg2、++ 参数将被插入到主字符串中的百分号（%）符号处。该函数是逐步执行的。在第一个 % 符号处，插入 arg1，在第二个 % 符号处，插入 arg2，依此类推

参数：format -- 必需。规定字符串以及如何格式化其中的变量

arg1 -- 必需。规定插到 format 字符串中第一个 % 符号处的参

arg2 -- 可选。规定插到 format 字符串中第二个 % 符号处的参数

arg++ -- 可选。规定插到 format 字符串中第三、四等等 % 符号处的参数

返回值：返回已格式化的字符串

**snprintf():**

描述：int snprintf ( char \* str, size\_t size, const char \* format, ... )

设将可变参数(...)按照 format 格式化成字符串，并将字符串复制到 str 中，size 为要写入的字符的最大数目，超过 size 会被截断

参数：str -- 目标字符串

size -- 拷贝字节数(Bytes)如果格式化后的字符串长度大于 size

format -- 格式化成字符串

返回值：如果格式化后的字符串长度小于等于 size，则会把字符串全部复制到 str 中，并给其后添加一个字符串结束符 \0。如果格式化后的字符串长度大于 size，超过 size 的部分会被截断，只将其中的 (size-1) 个字符复制到 str 中，并给其后添加一个字符串结束符 \0，返回值为欲写入的字符串长度

**vsprintf():PHP**

描述：vsprintf(format,argarray)

与 sprintf() 不同，vsprintf() 中的参数位于数组中。数组元素将被插入到主字符串中的百分号（%）符号处。该函数是逐步执行的

参数：format -- 必需。规定字符串以及如何格式化其中的变量

argarray -- 必需。带有参数的一个数组，这些参数会被插到 format 字符串中的 % 符号处

返回值：以格式化字符串的形式返回数组值

**vsnprintf():**

描述：int vsnprintf (char \* s, size\_t n, const char \* format, va\_list arg )

将格式化数据从可变参数列表写入大小缓冲区

如果在 printf 上使用格式，则使用相同的文本组成字符串，但使用由 arg 标识的变量参数列表中的元素而不是附加的函数参数，并将结果内容作为 C 字符串存储在 s 指向的缓冲区中（以 n 为最大缓冲区容量来填充）。如果结果字符串的长度超过了 n-1 个字符，则剩余的字符将被丢弃并且不被存储，而是被计算为函数返回的值。在内部，函数从 arg 标识的列表中检索参数，就好像 va\_arg 被使用了一样，因此 arg 的状态很可能被调用所改变。在任何情况下，arg 都应该在调用之前的某个时刻由 va\_start 初始化，并且在调用之后的某个时刻，预计将由 va\_end 释放

参数：s -- 指向存储结果 C 字符串的缓冲区的指针，缓冲区应至少有 n 个字符的大小

n -- 在缓冲区中使用的最大字节数，生成的字符串的长度至多为 n-1，为额外的终止空字符留下空，size\_t 是一个无符号整数类型

format -- 包含格式字符串的 C 字符串，其格式字符串与 printf 中的格式相同

arg -- 标识使用 va\_start 初始化的变量参数列表的值

返回值：如果 n 足够大，则会写入的字符数，不包括终止空字符。如果发生编码错误，则返回负数。注意，只有当这个返回值是非负值且小于 n 时，字符串才被完全写入

**gets():**

描述：char \*gets(char \*str)

从标准输入 stdin 读取一行，并把它存储在 str 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定

参数：str -- 这是指向一个字符数组的指针，该数组存储了 C 字符串

返回值：如果成功，该函数返回 str。如果发生错误或者到达文件末尾时还未读取任何字符，则返回 NULL

**GCC 用法:**

gcc -D\_FORTIFY\_SOURCE=1 仅仅只在编译时进行检查（尤其是#include <string.h>这种文件头）

gcc -D\_FORTIFY\_SOURCE=2 程序执行时也会进行检查（如果检查到缓冲区溢出，就会终止程序）

在-D\_FORTIFY\_SOURCE=2 时，通过对数组大小来判断替换 strcpy、memcpy、memset 等函数名，从而达到防止缓冲区溢出的作用

**pwntools 学习**

1、大致框架

from pwn import \* #用来导入 pwntools 模块

context(arch = 'i386', os = 'linux') #设置目标机的信息

r = remote('exploitme.example.com', 31337)

'''

用来建立一个远程连接，url 或者 ip 作为地址，然后指明端口

这里也可以仅仅使用本地文件,调试时方便:

'''

r = process("./test")

#test 即为文件名,这使得改变远程和本地十分方便.

asm(shellcraft.sh())

'''

asm()函数接收一个字符串作为参数，得到汇编码的机器代码。

比如

```
>>> asm('mov eax, 0')
```

```
'\xb8\x00\x00\x00\x00'
```

shellcraft 模块是 shellcode 的模块，包含一些生成 shellcode 的函数。

其中的子模块声明架构，比如 shellcraft.arm 是 ARM 架构的，shellcraft.amd64 是 AMD64 架构，shellcraft.i386 是 Intel 80386 架构的，以及有一个 shellcraft.common 是所有架构通用的。而这里的 shellcraft.sh()则是执行/bin/sh 的 shellcode 了

'''

```
r.send() #将 shellcode 发送到远程连接
```

```
r.interactive()
```

#将控制权交给用户，这样就可以使用

## 2、Context 设置

context 是 pwntools 用来设置环境的功能。在很多时候，由于二进制文件的情况不同，我们可能需要进行一些环境设置才能够正常运行 exp，比如有一些需要进行汇编，但是 32 的汇编和 64 的汇编不同，如果不设置 context 会导致一些问题。

一般来说我们设置 context 只需要简单的一句话：

```
context(os='linux', arch='amd64', log_level='debug')
```

这句话的意思是：

1. os 设置系统为 linux 系统，在完成 ctf 题目的时候，大多数 pwn 题目的系统都是 linux
2. arch 设置架构为 amd64，可以简单的认为设置为 64 位的模式，对应的 32 位模式是 'i386'
3. log\_level 设置日志输出的等级为 debug，这句话在调试的时候一般会设置，这样 pwntools 会将完整的 io 过程都打印下来，使得调试更加方便，可以避免在完成 CTF 题目时出现一些和 IO 相关的错误。

## 3、数据打包

数据打包,即将整数值转换为 32 位或者 64 位地址一样的表示方式,比如 0x400010 表示为 \x10\x00\x40 一样,这使得我们构造 payload 变得很方便

用法:

\* p32/p64: 打包一个整数,分别打包为 32 或 64 位

\* u32/u64: 解包一个字符串,得到整数

p 对应 pack,打包,u 对应 unpack,解包,简单好记

```
payload = p32(0xdeadbeef) # pack 32 bits number
```

## 4、数据输出

如果需要输出一些信息,最好使用 pwntools 自带的,因为和 pwntools 本来的格式吻合,看起来也比较舒服,用法:

```
some_str = "hello, world"
```

```
log.info(some_str)
```

其中的 info 代表是 log 等级，也可以使用其他 log 等级。

## 5、Cyclic Pattern

Cyclic pattern 是一个很强大的功能，大概意思就是，使用 pwntools 生成一个 pattern，pattern 就是指一个字符串，可以通过其中的一部分数据去定位到他在一个字符串中的位置。

在我们完成栈溢出题目的时候，使用 pattern 可以大大的减少计算溢出点的时间。

用法:

`cyclic(0x100)` # 生成一个 0x100 大小的 pattern，即一个特殊的字符串

`cyclic_find(0x61616161)` # 找到该数据在 pattern 中的位置（或者是 `cyclic -l 0x61616161`）

`cyclic_find('aaaa')` # 查找位置也可以使用字符串去定位

比如，我们在栈溢出的时候，首先构造 `cyclic(0x100)`，或者更长长度的 pattern，进行输入，输入后 pc 的值变为了 0x61616161，那么我们通过 `cyclic_find(0x61616161)` 就可以得到从哪一个字节开始会控制 PC 寄存器了，避免了很多没必要的计算。

## 6、汇编与 shellcode

有的时候我们需要在写 exp 的时候用到简单的 shellcode，pwntools 提供了对简单的 shellcode 的支持。

首先，常用的，也是最简单的 shellcode。

shellcraft : shellcode 的生成器。即调用 `/bin/sh` 可以通过 shellcraft 得到：

注意，由于各个平台，特别是 32 位和 64 位的 shellcode 不一样，所以最好先设置 context，如果没声明平则

32 位: `shellcraft.i386.linux.sh()`

64 位: `shellcraft.amd64.linux.sh()`

`print(shellcraft.sh())` # 打印出 shellcode

不过，现在看到的 shellcode 还是汇编代码，不是能用的机器码，所以还需要进行一次汇编

`print(asm(shellcraft.sh()))` # 打印出汇编后的 shellcode

asm 可以对汇编代码进行汇编，不过 pwntools 目前的 asm 实现还有一些缺陷，比如不能支持相对跳转等等，只可以进行简单的汇编操作。如果需要更复杂一些的汇编功能，可以使用 keystone-engine 项目，这里就不再赘述了。

asm 也是架构相关，所以一定要先设置 context，避免一些意想不到的错误。

asm 也是架构相关，所以一定要先设置 context，避免一些意想不到的错误。

## 7、ELF 文件操作

```
In [1]: from pwn import *
```

```
In [2]: elf = ELF('./level0')
```

```
[*] '/home/nuo/level0'
```

```
Arch:      amd64-64-little
```

```
RELRO:     No RELRO
```

```
Stack:     No canary found
```

```
NX:        NX enabled
```

```
PIE:       No PIE (0x400000)
```

```
In [3]: callsys_addr = elf.symbols['callsystem']
```

```
In [4]: print callsys_addr
```

```
4195734
```

```
In [6]: a=hex(callsys_addr)
```

```
In [7]: print a
```

```
0x400596
```

可见 ipython 时，ELF 相当于 checksec，但其主要是获取信息，一些地址等

```
>>> e = ELF('/bin/cat')
```

```
>>> print hex(e.address) # 文件装载的基地址
```

```
0x400000
```

```
>>> print hex(e.symbols['write']) # 函数地址,symbols,got,plt 均是列表
```

0x401680

>>> print hex(e.got['write']) # GOT 表的地址

0x60b070

>>> print hex(e.plt['write']) # PLT 的地址

0x401680

>>> print hex(e.search('/bin/sh').next())# 字符串/bin/sh 的地址字符串加 ( )

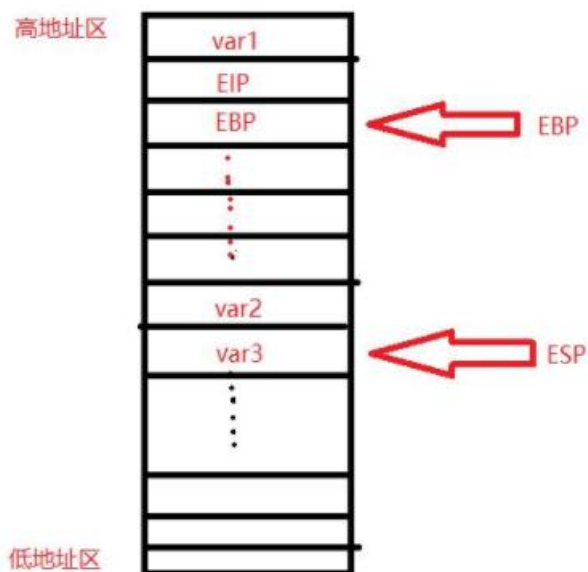
栈

### 栈的结构

- 1、先进后出。
- 2、在内存中表现为从高地址往低地址增长。
- 3、栈顶：栈的最上方（低地址区）。
- 4、栈底：栈的最下方（高地址区）。

```
void fun(int a) {  
  
    int b;  
  
    char s;  
  
    gets(&s);  
  
    if(a == 0x1234){  
  
        puts(&s);  
  
    }  
  
}
```

下面是这个栈的结构图：



从栈的结构中我们可以看到，这个栈中周多个临时变量，数字的代表其进站顺序。其中 ESP 指向了 var3，在栈的顶部，EBP 指向了栈的底部。在 EBP 的下面还有一个 EIP,这里其实可以理解为我们的函数返回地址。当 return 语句执行后，下一条指令的执行地址。那么 var1 是什么呢？其实这个是函数的参数，我们对对应代码说明一下 var1-var3。

var1: 参数 a

var2: 变量 b

var3: 变量 s

题目

攻防世界 pwn 新手

1, 2, 3

栈溢出

[https://mp.weixin.qq.com/s?\\_\\_biz=MzkzMzI2NTg5NA==&mid=2247483660&idx=1&sn=0be9f09f8391af9833d6122c30290107&chksm=c24e6985f539e093a0dc143f899040e9dd660ca2c9e5f5aa54a717893fd37056ad1c091a4cfd&token=604422757&lang=zh\\_CN#rd](https://mp.weixin.qq.com/s?__biz=MzkzMzI2NTg5NA==&mid=2247483660&idx=1&sn=0be9f09f8391af9833d6122c30290107&chksm=c24e6985f539e093a0dc143f899040e9dd660ca2c9e5f5aa54a717893fd37056ad1c091a4cfd&token=604422757&lang=zh_CN#rd)

```
>>> int('0x88',16)
136
>>> int('0x20',16)
32
>>> █
```