pwn复习

汇编

寄存器

`eax ebx ecx edx esi edi`

`rax rbx rcx rdx rsi rdi`

`ebp/rbp` 基指针

`esp/rbp` 栈指针

`eip/rip` -> 不能用`mov/pop/push`直接操作， `jmp...` `call` `ret`

数据传送指令

`mov`

地址传送指令

`lea`

栈操作指令

`pop` 弹出

`push` 压入

跳转指令

过程调用

`call addr` 相当于 `push eip; jmp addr;`

`ret` 相当于 `pop eip;`

跳转

`jmp` 无条件跳转，操作`eip/rip`

`jz` `jnz` ..... 条件跳转

调用约定

建议自己编译然后用pwndbg调试看一下具体栈是怎么在函数调用中变化的

```c
int func(int i, int j)
{
    return i + j;
}

int main(int argc, char const *argv[])
{
    int i;
    i = func(1, 2);
    return 0;
}
```

32位

参数都通过栈传递，从最右侧开始向左依次入栈，最左侧的参数地址最低，最靠近返回地址的位置，最右侧的参数地址最高

```
main:
push    ebp
mov     ebp, esp
sub     esp, 10h
push    2
push    1
call    func
add     esp, 8
mov     [ebp+var_4], eax
mov     eax, 0
leave
retn

func:
push    ebp
mov     ebp, esp
mov     edx, [ebp+arg_0]
mov     eax, [ebp+arg_4]
add     eax, edx
pop     ebp
retn
```

64位

整型参数从左向右依次放在 `rdi rsi rdx rcx r8 r9`

浮点参数从左向右依次放在 `xmm0 ~ xmm7`

更多的参数通过栈传递

```
main:
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_14], edi
mov     [rbp+var_20], rsi
mov     esi, 2
mov     edi, 1
call    func
```

```
mov     [rbp+var_4], eax
mov     eax, 0
leave
retn

func:
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     edx, [rbp+var_4]
mov     eax, [rbp+var_8]
add     eax, edx
pop     rbp
retn
```

函数序言和函数尾声

```
push ebp
mov ebp,esp
sub esp,X
```

```
mov esp,ebp
pop ebp
ret
```

`enter` -> `push ebp; mov ebp,esp;`

`leave` -> `mov esp,ebp; pop ebp;`

Linux上的安全保护

ASLR

系统级别的保护，开启后可以使程序每次运行时随机化栈、堆的基地址和运行库加载的基地址，如果程序是PIE，程序本身的基地址也随机化。常见的Linux系统都开启了此保护。

NX

程序级别，编译时决定，数据区域（包括全局数据，堆，栈）不可执行

PIE

程序级别，编译时决定，开启后编译出的程序为位置无关可执行文件，可以被ASLR随机化。

stack protector

程序级别，编译时决定，开启后编译出的程序中的函数会向栈底附近插入随机值Canary，返回时会检查是否被修改。

fortify

程序级别，编译时决定，开启后风险库函数（如`str*, mem*`）会加入边界检查

relro

程序级别，编译时决定，有多个级别，partial relro时.got.plt可写，full relro时整个got只读

栈溢出

通过越界写局部变量，覆盖栈中函数的返回地址以控制程序控制流

ROP

因为开启了上述的保护，现在一般无法得到已知地址的可读可写可执行（RWX）的内存区域，所以难以写入shellcode。因此尝试通过程序中已有的指令来进行攻击。因为包含ret指令的小片段（gadget）方便进行组合（在栈上布置这些片段的地址，然后在片段中的这些`ret`的驱动下依次执行这些片段），因此来实现一定的指令组合达到攻击目的。

ret to plt

GOT

全局偏移表，外部符号（比如库函数）的实际偏移（地址）表

PLT

PLT（Procedure Linkage Table）程序链接表。它有两个功能，要么在 `.got.plt` 节中拿到地址，并跳转。

要么当 `.got.plt` 没有所需地址的时，触发「动态链接器」去找到所需地址

`.got.plt`是GOT的一部分，GOT的库函数部分单独分出来就是`.got.plt`

调用

plt中出现的函数，直接跳到plt的地址就能调用，plt相当于一个跳板

ret to libc

plt中没有的函数，就需要去libc中去找函数入口的真实加载地址然后调用。因为libc加载地址随机，所以需要获取libc的加载的基地址。

一般来说会给出libc文件，这样就能知道各个函数距离libc基地址的偏移（地址差）。然后拿到一个函数的真实地址。

例如已知

```
printf_offset = 0x12345
system_offset = 0x23333
printf_real=0x7fffdfdf1345
```

那么

```
libc_base = printf_real - printf_offset  = 0x7fffdfddf000
system_real = liobc_base + system_offset = 0x7fffdfe02333
```

注意无论32位64位，函数地址的低12位（最后3个数字）不会被随机化，所以可以从此初步判断泄露的地址是不是正确。计算出来的libc基地址最后3个数字都应该是0