## 0x00 序言

[上一篇](http://www.freebuf.com/vuls/179724.html)谈到的栈溢出攻击方法是 **ret2shellcode**，其主要思想就是控制返回地址使其指向 **shellcode**所在的区域 。该技术能够成功的关键点在于： 1、程序存在溢出，并且还要能够控制返回地址  2、程序运行时，shellcode 所在的区域要拥有执行权限  3、操作系统还需要关闭  ASLR (地址空间布局随机化)  保护 。 现在开启  **DEP(Data Execution Prevention)/NX(Non-executable)  数据执行保护**，通过利用 **ret2libc**技术绕过该保护机制，接下来就是通过示例演示如何利用 **ret2libc**的攻击方式实现任意代码执行。

## 0x01 利用思路

**ret2libc**这种攻击方式主要是针对 **动态链接(Dynamic linking)** 编译的程序，因为正常情况下是无法在程序中找到像 **system() 、execve()** 这种系统级函数(如果程序中直接包含了这种函数就可以直接控制返回地址指向他们，而不用通过这种麻烦的方式)。因为程序是动态链接生成的，所以在程序运行时会调用**libc.so (程序被装载时，动态链接器会将程序所有所需的动态链接库加载至进程空间，libc.so 就是其中最基本的一个)**，**libc.so**是 linux 下 C 语言库中的运行库 **glibc**的动态链接版，并且 **libc**.**so**中包含了大量的可以利用的函数，包括 **system() 、execve()** 等系统级函数，我们可以通过找到这些函数在内存中的地址覆盖掉返回地址来获得当前进程的控制权。通常情况下，我们会选择执行 **system("/bin/sh")**来打开 shell， 如此就只剩下两个问题：1、找到 system() 函数的地址  2、在内存中找到 "/bin/sh" 这个字符串的地址

## 0x02 什么是动态链接（Dynamic linking）

**动态链接** 是指在程序装载时通过 **动态链接器**将程序所需的所有 **动态链接库(Dynamic linking library)** 装载至进程空间中（程序按照模块拆分成各个相对独立的部分），当程序运行时才将他们链接在一起形成一个完整程序的过程。它诞生的最主要的的原因就是 **静态链接**太过于浪费内存和磁盘的空间，并且现在的软件开发都是模块化开发，不同的模块都是由不同的厂家开发，在 **静态链接**的情况下，一旦其中某一模块发生改变就会导致整个软件都需要重新编译，而通过 **动态链接**的方式就推迟这个链接过程到了程序运行时进行。这样做有以下几点好处：

### 1、节省内存、磁盘空间

     例如磁盘中有两个程序，p1、p2，且他们两个都包含 lib.o 这个模块，在 **静态链接**的情况下他们在链接输出可执行文件时都会包含 lib.o 这个模块，这就造成了磁盘空间的浪费。当这两个程序运行时，内存中同样也就包含了这两个相同的模块，这也就使得内存空间被浪费。当系统中包含大量类似 lib.o 这种被多个程序共享的模块时，也就会造成很大空间的浪费。在 **动态链接**的情况下，运行 p1 ，当系统发现需要用到 lib.o ，就会接着加载 lib.o 。这时我们运行 p2 ，就不需要重新加载 lib.o 了，因为此时 lib.o 已经在内存中了，系统仅需将两者链接起来，此时内存中就只有一个 lib.o 节省了内存空间。

### 2、程序更新更简单

     比如程序 p1 所使用的 lib.o 是由第三方提供的，等到第三方更新、或者为 lib.o 打补丁的时候，p1 就需要拿到第三方最新更新的 lib.o ，重新链接后在将其发布给用户。程序依赖的模块越多，就越发显得不方便，毕竟都是从网络上获取新资源。在 **动态链接**的情况下，第三方更新 lib.o 后，理论上只需要覆盖掉原有的 lib.o ，就不必重新链接整个程序，在程序下一次运行时，新版本的目标文件就会自动装载到内存并且链接起来，就完成了升级的目标。

### 3、增强程序扩展性和兼容性

**动态链接** 的程序在运行时可以动态地选择加载各种模块，也就是我们常常使用的插件。软件的开发商开发某个产品时会按照一定的规则制定好程序的接口，其他开发者就可以通过这种接口来编写符合要求的动态链接文件，以此来实现程序功能的扩展。增强兼容性是表现在 **动态链接**的程序对不同平台的依赖差异性降低，比如对某个函数的实现机制不同，如果是 **静态链接**的程序会为不同平台发布不同的版本，而在 **动态链接**的情况下，只要不同的平台都能提供一个动态链接库包含该函数且接口相同，就只需用一个版本了。

    总而言之，**动态链接**的程序在运行时会根据自己所依赖的 **动态链接库**，通过 **动态链接器**将他们加载至内存中，并在此时将他们链接成一个完整的程序。Linux 系统中，**ELF**动态链接文件被称为 **动态共享对象（Dynamic Shared Objects）** ， 简称 **共享对象**一般都是以 “.so” 为扩展名的文件；在 windows 系统中就是常常软件报错缺少 xxx.dll 文件。

## 0x03 GOT (Global offset Table)

   了解完 **动态链接**，会有一个问题：**共享对象**在被装载时，如何确定其在内存中的地址？下面简单的介绍一下，要使 **共享对象**能在任意地址装载就需要利用到 **装载时重定位**的思想，即在链接时对所有的绝对地址的引用不做重定位而将这一步推迟到装载时再完成，一旦装载模块确定，系统就对所有的绝对地址引用进行重定位。但是随之而来的问题是，指令部分无法在多个进程之间共享，这又产生了一个新的技术 **地址无关代码 （PIC，Position-independent Code）**，该技术基本思想就是将指令中需要被修改的部分分离出来放在数据部分，这样就能保证指令部分不变且数据部分又可以在进程空间中保留一个副本，也就避免了不能节省空间的情况。那么重新定位后的程序是怎么进行数据访问和函数调用的呢？下面用实际代码验证 :

got\_extern.h

int b;

got.c

#include <stdio.h>

#include "got\_extern.h"

static int a;

extern int b;

int fun()

{

a = 1;

b = 2;

}

int main(int argc, char const \*argv[])

{

fun();

printf("hey!");

return 0;

}

编译成32位程序

gcc got.c -m32 -o got

用 objdump 查看反汇编代码 objdump -D -Mintel got

000011a9 <fun>:

11a9: 55 push ebp

11aa: 89 e5 mov ebp,esp

11ac: e8 5e 00 00 00 call 120f <\_\_x86.get\_pc\_thunk.ax>

11b1: 05 4f 2e 00 00 add eax,0x2e4f

11b6: c7 80 20 00 00 00 01 mov DWORD PTR [eax+0x20],0x1

11bd: 00 00 00

11c0: 8d 80 24 00 00 00 lea eax,[eax+0x24]

11c6: c7 00 02 00 00 00 mov DWORD PTR [eax],0x2

11cc: 90 nop

11cd: 5d pop ebp

11ce: c3 ret

000011cf <main>:

11cf: 8d 4c 24 04 lea ecx,[esp+0x4]

11d3: 83 e4 f0 and esp,0xfffffff0

11d6: ff 71 fc push DWORD PTR [ecx-0x4]

11d9: 55 push ebp

11da: 89 e5 mov ebp,esp

11dc: 53 push ebx

11dd: 51 push ecx

11de: e8 cd fe ff ff call 10b0 <\_\_x86.get\_pc\_thunk.bx>

11e3: 81 c3 1d 2e 00 00 add ebx,0x2e1d

11e9: e8 bb ff ff ff call 11a9 <fun>

11ee: 83 ec 0c sub esp,0xc

11f1: 8d 83 08 e0 ff ff lea eax,[ebx-0x1ff8]

11f7: 50 push eax

11f8: e8 43 fe ff ff call 1040 <printf@plt>

11fd: 83 c4 10 add esp,0x10

1200: b8 00 00 00 00 mov eax,0x0

1205: 8d 65 f8 lea esp,[ebp-0x8]

1208: 59 pop ecx

1209: 5b pop ebx

120a: 5d pop ebp

120b: 8d 61 fc lea esp,[ecx-0x4]

120e: c3 ret

**1、模块内部调用**

main()函数中调用 fun()函数 ，指令为

11e9: e8 bb ff ff ff call 11a9 <fun>

fun() 函数所在的地址为 0x000011a9 ，机器码 e8 代表 call 指令，为什么后面是 bb ff ff ff 而不是 a9 11 00 00 （小端存储）呢？这后面的四个字节代表着目的地址相对于当前指令的下一条指令的偏移，即 0x11e9 + 0x5 + (-69) = 0x11a9 ，0xffffffbb 是 -69 的补码形式，这样做就可以使程序无论被转载到哪里都会正常执行。

### 2、模块内部数据访问

ELF 文件是由很多很多的 **段(segment)** 所组成，常见的就如  .text (代码段) 、.data(数据段，存放已经初始化的全局变量或静态变量)、.bss(数据段，存放未初始化全局变量)等，这样就能做到数据与指令分离互不干扰。在同一个模块中，一般前面的内存区域存放着代码后面的区域存放着数据(这里指的是 .data 段)。那么指令是如何访问远在 .data 段 中的数据呢?  
观察 fun() 函数中给静态变量 a 赋值的指令

