模块一：基本会出现漏洞的函数

ssize\_t **read** (int fd, void \*buf, size count);

int fd是文件指针。

void \*buf将读上来的数据保存到缓冲区buf里面，同时文件的当前读写位置向后移动。 size是请求读取的字节数。

ssize\_t **write**(int fd, const void\*buf, size count);

int fd是文件描述符（write所对应的是写，即就是1）。

buf是需要写入的字符串。

count是写入的字节数。

模块二：常用的命令和技巧

1. gcc -fno-stack-protector -z execstack -o XXX XXX.c

用gcc编译程序。-fno-stack-protector和-z execstack这两个参数会分别关掉dep和stack protector。

并且我们还要关掉linux里面的ASLR保护，在终端输入：

sudo -s

echo 0>/proc/sys/kernel/randomize\_va\_space

#表示把0送入到这个文件里面，为暂时修改，重启自动还原#

exit

1. 可以用gdb插件里面的cyclic XX 创建XX个字符

用cyclic -l 0x某某地址 🡪来判断用了多少个字符溢出

可以用patteern.py脚本来制造XX个字符 python pattern.py create XX

用python pattern.py offset 0x某某地址 🡪来判断

1. 一段最简单的执行execve ("/bin/sh")命令以<execve开启一个bash进程，调用路径为/bin/sh>的语句作为shellcode。把它编译成二进制数据，然后输入到我们能够输入的地方，如果NX没有开启的话，把返回地址覆盖后指向该shellcode就可以拿到shell了。
2. 值得注意的地方，在gdb调试目标程序时，查看内存来确定shellcode的位置，但是去运行exp的时候，shellcode不在那个地址上。因为gdb调试环境会影响buf在内存中的位置，即使我们关闭了ASLR，但是也只能保证buf的地址在gdb调试环境中不变，去真正执行./XXX的时候，buf的位置会固定在别的位置上。为了解决这个问题我们可以开启core dump。

ulimit -c unlimited

sudo sh -c ‘echo “/tmp/core.%t”>/proc/sys/kernel/core\_pattern’

开启之后，当出现内存错误的时候，系统会生成一个core dump文件在tmp目录下.然后我们再用gdb查看这个core文件就可以获取到buf真正的地址了。

1. 我们把这个目标程序作为一个服务绑定到服务器的某个端口上，这里我们可以使用socat这个工具来完成，命令如下:socat TCP4-LISTEN:10001,fork EXEC:./XXX

随后这个程序的io就被重定向到10001这个端口上了，并且可以使用 nc 127.0.0.1 10001来访问我们的目标程序服务了。因为现在目标程序是跑在socat的环境中，exp脚本除了要把p = process('./XXX')换成p = remote('127.0.0.1',10001) 之外，ret的地址还会发生改变。解决方法还是采用生成core dump的方案，然后用gdb调试core文件获取返回地址.然后我们就可以使用exp进行远程溢出啦!

1. 我们首先在main函数上下一个断点，然后执行程序，这样的话程序会加载libc.so到内存中，然后我们就可以通过”print system”这个命令来获取system函数在内存中的位置，随后我们可以通过” print \_\_libc\_start\_main”这个命令来获取libc.so在内存中的起始位置。

然后用find 查找/bin/sh这个字符串。这样做是为了得到system函数的地址和/bin/sh字符串的地址。因为单单system(Command)是无法得到shell的，必须要/bin/sh这个字符串。

Payload的构建如下：

<输入的数据>🡪<EBP>🡪<返回地址>🡪<被调函数里面的返回地址>🡪<被调函数的参数>

所以拿到shell的思路为：填充无用字符串，覆盖原来的返回地址到system函数，system函数后面跟的是执行完system函数后要返回地址，后面是system函数的参数，也就是/bin/sh这个字符串地址，因为我们拿到shell后不需要返回了，system函数的地址就可以乱写了。

1. 开启ASLR的保护后的思路

sudo -s

echo 2>/proc/sys/kernel/randomize\_va\_space

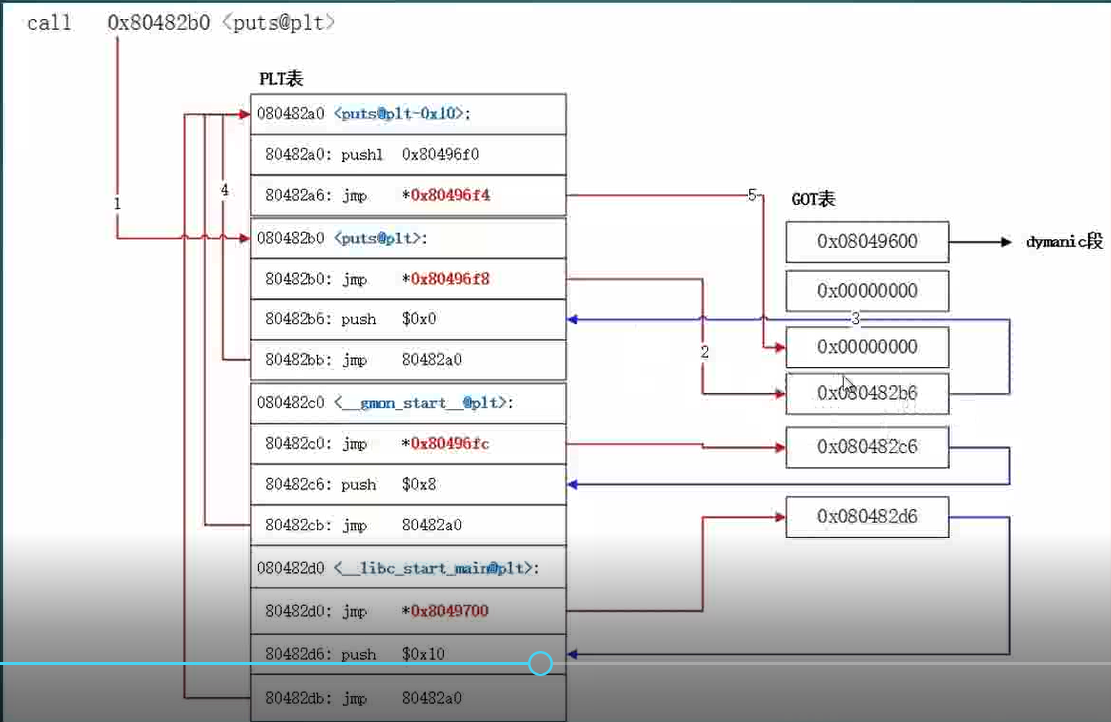
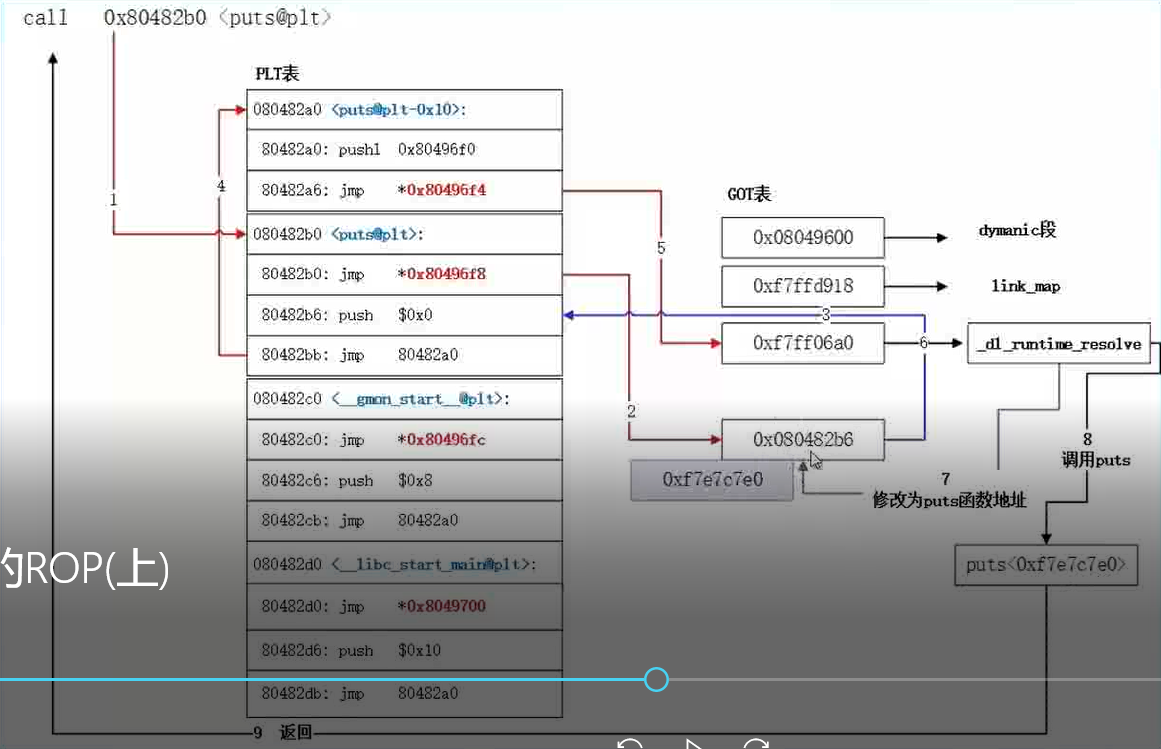
之后的话。通过sudo cat /proc/[pid]/maps或者ldd查看，你会发现libc.so地址每次都是变化的。解决地址随机化的问题是一步到位，不让它有机会运行第二次，在一次中地址是固定的，只是我们不知道而已。我们需要先泄漏出libc.so某些函数在内存中的地址，然后再利用泄漏出的函数地址根据偏移量计算出system()函数和/bin/sh字符串在内存中的地址，然后再执行我们的shellcode。<这里会涉及到libc.so版本的问题，有点题目需要先泄露出lib.c的版本再泄露函数，然后再泄露system的偏移>

1. 使用objdump来查看可以利用的plt函数以及函数对应的got表。

objdump -d -j .plt XXX(XXX为elf文件)

objdump -R XXX (查看got表)

1. 在程序本身没有调用system()时，我们需要借助打印函数来帮下我们我们可以用write()也就是把write.got打印出来。因为write函数存在于libc.so中。然后linux采用了延时绑定技术，我们去调用write@plt()的时候，首先jmp到GOT表的write函数中。之后再跳转到之前write@plt() jmp指令的下一条 push ￥0x某某 ，某某为write在plt表的序号，然后jmp到了plt表的第一项，这里是一个全局函数，push一个值后。会jmp到GOT表的第三项，暂时是0x00000000，实际上在运行的时候，ld加载我们的程序时，会去修正这个值，这个值就换成了运行时重定位中很重要的\_dl\_runtime\_resolve这个函数的地址，这个\_dl\_runtime\_resolve函数执行时，会修正GOT表里面的值，改为运行时write函数的地址，然后调用write函数在libc.so的地址，调用地址就是调用函数。GOT表第一项是dymanic段的地址，第二项是link\_map的地址。并且只有程序调用的库函数才会出现在plt表中。

system()函数和write()在libc.so中的offset(相对地址)是不变的，所以如果我们得到了write()的地址并且拥有目标服务器上的libc.so(就是搞清楚libc的版本)就可以计算出system()在内存中的地址了。

我们要使用ldd命令可以查看目标程序调用的so库。

ldd XXX

随后我们把libc.so拷贝到当前目录，因为我们的exp需要这个so文件来计算相对地址。

cp /lib/i386-linuc-gnu/libc.so.6 libc.so <举例>

模块三：其他知识点

1.Linux 中的 shell 有很多类型，其中最常用的几种是: Bourne shell (sh)、C shell (csh) 和 Korn shell (ksh), 各有优缺点。Bourne shell 是 UNIX 最初使用的 shell，并且在每种 UNIX 上都可以使用, 在 shell 编程方面相当优秀，但在处理与用户的交互方面做得不如其他几种shell。Linux 操作系统缺省的 shell 是Bourne Again shell，它是 Bourne shell 的扩展，简称 Bash，与 Bourne shell 完全向后兼容，并且在Bourne shell 的基础上增加、增强了很多特性。Bash放在/bin/bash中，它有许多特色，可以提供如命令补全、命令编辑和命令历史表等功能，它还包含了很多 C shell 和 Korn shell 中的优点，有灵活和强大的编程接口，同时又有很友好的用户界面。

GNU/Linux 操作系统中的 /bin/sh 本是 bash (Bourne-Again Shell) 的符号链接，但鉴于 bash 过于复杂，有人把 ash 从 NetBSD 移植到 Linux 并更名为 dash (Debian Almquist Shell)，并建议将 /bin/sh 指向它，以获得更快的脚本执行速度。Dash Shell 比 Bash Shell 小的多，符合POSIX标准。应该说, /bin/sh 与 /bin/bash 虽然大体上没什么区别, 但仍存在不同的标准. 标记为 “#!/bin/sh” 的脚本不应使用任何 POSIX 没有规定的特性 (如 let 等命令, 但 “#!/bin/bash” 可以). Debian 曾经采用 /bin/bash 更改 /bin/dash，目的使用更少的磁盘空间、提供较少的功能、获取更快的速度。但是后来经过 shell 脚本测试存在运行问题。因为原先在 bash shell 下可以运行的 shell script (shell 脚本)，在 /bin/sh 下还是会出现一些意想不到的问题，不是100%的兼用。

2. 