ret2libc

在ret2text那一篇文档中，我们已经大致了解了Linux系统的虚拟地址空间的结构。其中默认拥有可执行权限的不只是text段的内容，还有位于中间的动态链接库的内容。在众多链接库中libc库就是最常用的libc库之一，在写c语言程序的过程中常用的函数如`printf`，`scanf`，`gets`，`puts`，`system`等函数都是实现在libc中。也就是说，我们将返回地址改成libc中的函数来get shell也是可行的。

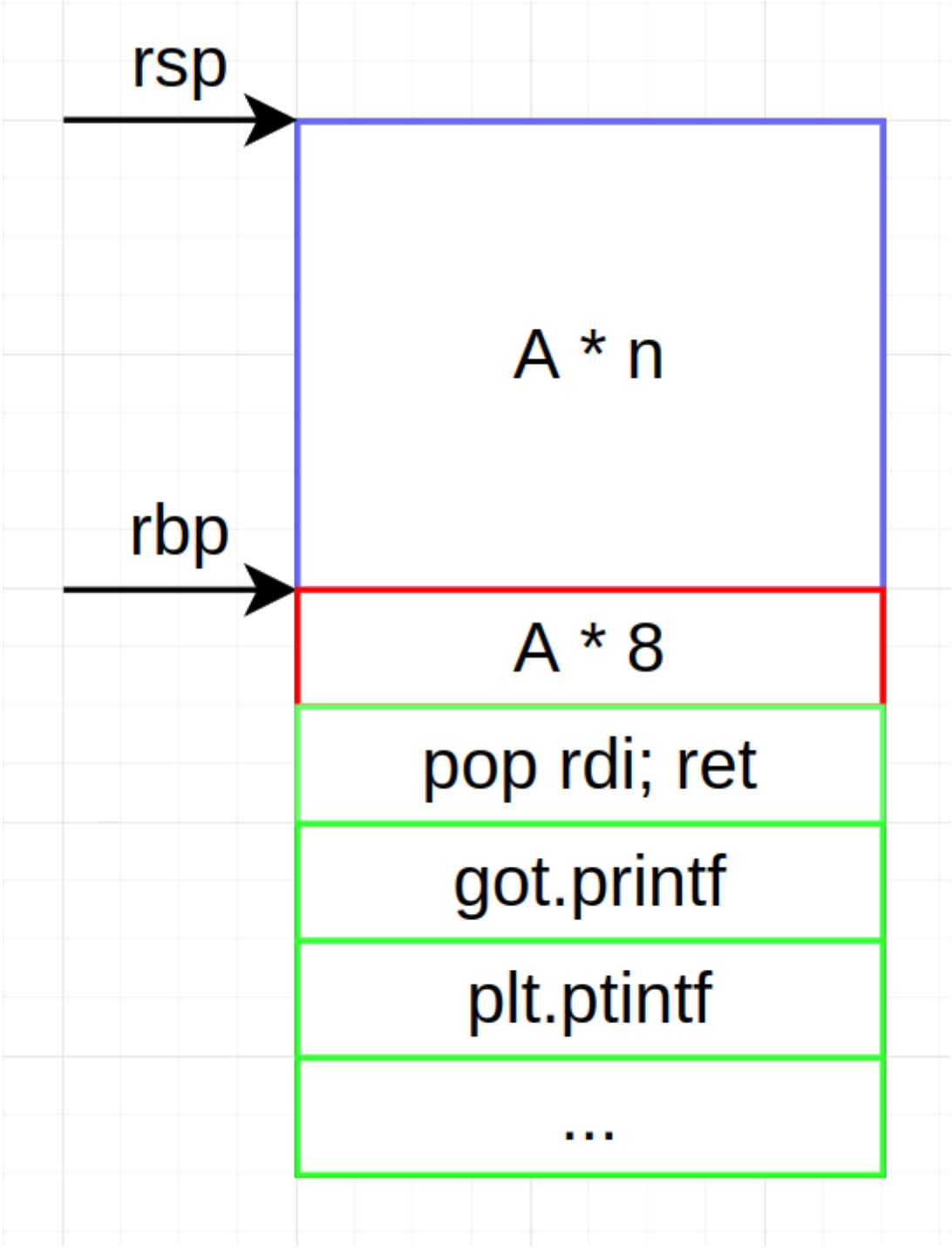
泄漏libc基地址

由于libc的ELF文件默认开启了地址随机化保护（这个保护具体的其他细节会在后续的文档中介绍），并且是动态加载，所以每次启动程序时，同一个函数的地址都有可能是不同的。但是ELF文件和内存之间依然存在映射关系，并且地址随机化保护仅仅使整个映射的基地址随机化，而所有内容相对于基地址的偏移不变。利用这个特性，我们就可以通过泄漏某个函数的地址，再通过libc的ELF文件获取对应的偏移，从而计算出libc在内存中映射的基地址。在获取到基地址后，我们就可以再次通过偏移计算出其他函数的地址，从而控制程序返回到libc中的函数中（text段中没有使用，但是在动态链接库中的函数）。

泄漏libc中的函数的地址

在Linux环境中动态链接库存在一个lazy load机制，即函数在初次被调用时才会进行链接，而这个机制通过`got`和`plt`两个表实现。`plt`表是一组函数，可以直接调用，`got`表是一组函数地址，记录了动态链接库中的函数的地址，初值为NULL，对应函数被调用过一次之后会变更为实际的函数地址。text段中的call动态链接库实际call的就是`plt`表中的函数，`plt`表首先会检查`got`表对应位置的值，若为NULL则调用链接函数，获取到函数地址，存到对应的`got`表中，并转调执行目标函数；若不为NULL，则直接按照`got`表转调到目标函数。

在攻击过程中泄漏libc地址就可以通过`got`和`plt`实现，原理很简单，只需要构造一个调用链，实现调用如`plt@ptintf(got@printf)`这样就能将`printf`函数的地址输出出来。在64位程序的ROP链中，就是构造类似如下的ROP链：



在泄漏出libc地址之后，再通过libc文件获取`printf`函数的偏移，实际地址减去偏移就是libc库映射到内存中的基地址。

重返漏洞函数

构造ROP时你会发现，第一次构造ROP链时，只能做到泄漏，而无法做到攻击，这时我们就需要在上面的ROP链中再次加上存在漏洞的函数，这样就可以再一次拿到栈溢出，进行第二次ROP，在这一次栈溢出中构造get shell的ROP链，最后完成攻击。

相关工具用法

pwntools

pwntools接收leak内容

```
from pwn import *

p = process("./vuln")

recv_bytes = p.recv()
```

相关接收函数有

```
recv(int: length)           # 接收指定长度的内容
recvuntil(bytes: str)       # 一直接收到指定字符串
```

解包函数，与p64，p32函数相对

```
u32()
u64()
```

pwntools这个库有加载libc库的及地址的功能，使用例子如下：

```
from pwn import *

libc = ELF("./libc.so")

printf_addr = 0xaaaaaaaa        # 假设leak了printf的地址为0xaaaaaaaa

libc_base = printf_addr - libc.sym["printf"]

system_addr = libc_base + libc.sym["system"]

binsh_addr = libc_base + next(libc.search(b"/bin/sh"))  # 搜索ELF中的字符串（注：glibc文件中存在字符串/bin/sh，需要时可以拿来利用）
```

patchelf

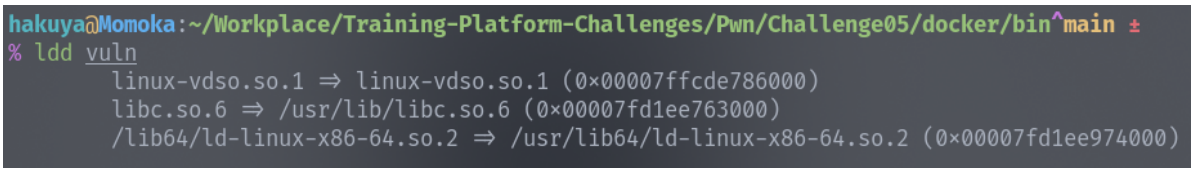
题目远程环境的libc版本和本地的libc并不一定相同，但是附件中包含了对应的libc文件，这时就需要我们自己手动替换libc文件。一般情况下，我们需要替换的为`ld.so`和`libc.so`连个文件，替换命令如下

```
patchelf --set-interpreter <new ld file> <ELF file>
patchelf --replace-needed <old libc> <new libc> <ELF file>
```

另外我们可以通过ldd命令查看ELF的链接信息

```
ldd <ELF file>
```

运行输出如下（patchelf替换libc命令中old libc指代的就是下图中的libc.so.6）



LibcSearcher

有时程序并不会提供libc文件，但是攻击时又涉及了libc，这里就可以用清华大佬开发的LibcSearcher，大致的使用方法如下：

```
from LibcSearcher import *

printf_addr = 0xaaaaaaaa        # 假设leak了printf的地址为0xaaaaaaaa

libc = LibcSearcher("printf", printf_addr)

libc_base = printf_addr - libc.dump("printf")

system_addr = libc_base + libc.dump("system")
binsh_addr = libc_base + libc.dump("str_bin_sh")
```

不过仅仅一个函数地址可能会搜索出来多个libc版本，虽然可以一个一个尝试，但并不是很方便。这时可以利用add_condition函数添加限制条件。例子如下：

```
from LibcSearcher import *

printf_addr = 0xaaaaaaaa        # 假设leak了printf的地址为0xaaaaaaaa
gets_addr = 0xbbbbbbbb      # 假设leak了gets的地址为0xbbbbbbbb

libc = LibcSearcher("printf", printf_addr)
libc.add_condition("gets", gets_addr)

libc_base = printf_addr - libc.dump("printf")

system_addr = libc_base + libc.dump("system")
binsh_addr = libc_base + libc.dump("str_bin_sh")
```

一些注意点

Linux的64位程序地址只用了48位，即6个字节，使用recv函数接收时，注意接收长度以及补位。

有时题目只提供了libc文件，但是ld和libc的版本有一定的关联，可能用本机的ld无法正常运行，又或是题目给的glibc没有符号表，这时可以用glibc-all-in-one这个工具获取对应版本的libc，并将链接库链接过去。

pwn题中常用的libc查看版本方法：

```
strings <libc file> | grep ubuntu
```

例如下图，`2.31-0ubuntu9.9`就是版本号

```
hakuya@Momoka:~/CTF/glibc-all-in-one/libs/2.31-0ubuntu9.9_amd64^master ±
% strings libc-2.31.so | grep "ubuntu"
GNU C Library (Ubuntu GLIBC 2.31-0ubuntu9.9) stable release version 2.31.
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

新版LibcSearcher需要联网使用。