APP漏洞扫描器之未使用地址空间随机化

阿里聚安全 / 2016-12-08 02:21:40 / 浏览数 3015 安全技术 漏洞分析 顶(0) 踩(0)

APP漏洞扫描用地址空间随机化

前言

我们在前文《APP漏洞扫描器之本地拒绝服务检测详解》了解到阿里聚安全漏洞扫描器有一项静态分析加动态模糊测试的方法来检测的功能，并详细的介绍了它在针对本地拒

同时，阿里聚漏洞扫描器有一个检测项叫未使用地址空间随机化技术,
该检测项会分析APP中包含的ELF文件判断它们是否使用了该项技术。如果APP中存在该项漏洞则会降低缓冲区溢出攻击的门槛。

本文主要介绍该项技术的原理和扫描器的检测方法。由于PIE的实现细节较复杂，本文只是介绍了大致的原理。想深入了解细节的同学可以参看潘爱民老师的书籍《程序员的

PIE是什么

PIE(position-independent
executable)是一种生成地址无关可执行程序的技术。如果编译器在生成可执行程序的过程中使用了PIE，那么当可执行程序被加载到内存中时其加载地址存在不可预知性。

PIE还有个孪生兄弟PIC(position-independent
code)。其作用和PIE相同，都是使被编译后的程序能够随机的加载到某个内存地址。区别在于PIC是在生成动态链接库时使用(Linux中的so)，PIE是在生成可执行文件时使用

PIE的作用

安全性

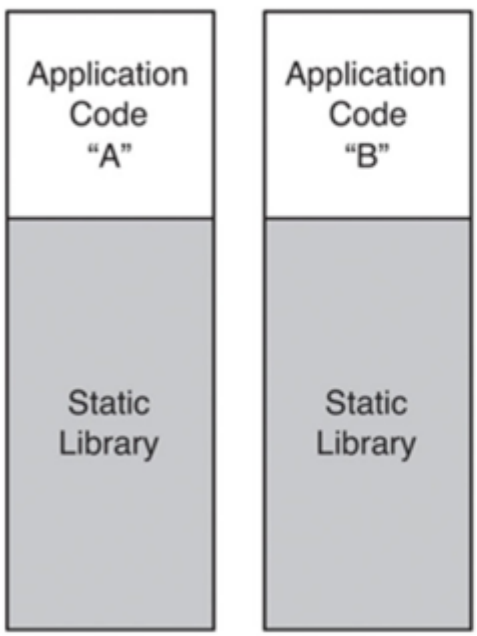
PIE可以提高缓冲区溢出攻击的门槛。它属于ASLR(Address space layout
randomization)的一部分。ASLR要求执行程序被加载到内存时，它其中的任意部分都是随机的。包括
Stack, Heap ,Libs and mmap, Executable, Linker, VDSO。通过PIE我们能够实现Executable 内存随机化

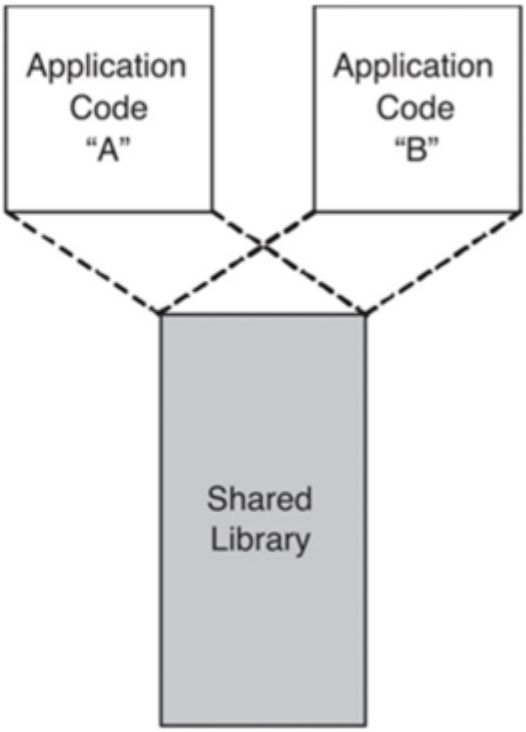
节约内存使用空间

除了安全性，地址无关代码还有一个重要的作用是提高内存使用效率。

一个共享库可以同时被多个进程装载，如果不是地址无关代码(代码段中存在绝对地址引用)，每个进程必须结合其自生的内存地址调用动态链接库。导致不得不将共享库整体

相反如果被加载的共享库是地址无关代码，100个进程调用该库，则该库只需要在内存中加载一次。这是因为PIE将共享库中代码段须要变换的内容分离到数据段。使得代码段





PIE工作原理简介

我们先从实际的例子出发，观察PIE和NO-PIE在可执行程序表现形式上的区别。管中窥豹探索地址无关代码的实现原理。

例子一

定义如下C代码：

```
#include <stdio.h>

int global;

void main()
{
printf("global address = %x\n", &global);
}
```

程序中定义了一个全局变量global并打印其地址。我们先用普通的方式编译程序。

```
gcc -o sample1 sample1.c
```

运行程序可以观察到global加载到内存的地址每次都一样。

```
$./sample1
global address = 6008a8
$./sample1
global address = 6008a8
$./sample1
global address = 6008a8
```

接着用PIE方式编译 sample1.c

```
gcc -o sample1_pie sample1.c -fpie -pie
```

运行程序观察global的输出结果：

```
./sample1_pie
global address = 1ce72b38
./sample1_pie
global address = 4c0b38
./sample1_pie
global address = 766dcb38
```

每次运行地址都会发生变换，说明PIE使执行程序每次加载到内存的地址都是随机的。

例子二

在代码中声明一个外部变量global。但这个变量的定义并未包含进编译文件中。

```
#include <stdio.h>

extern int global;

void main()
{
printf("extern global address = %x\n", &global);
}
```

首先使用普通方式编译 extern_var.c。在编译选项中故意不包含有global定义的源文件。

```
gcc -o extern_var extern_var.c
```

发现不能编译通过, gcc提示：

```
/tmp/ccJYN5Ql.o: In function `main':
extern_var.c:(.text+0xa): undefined reference to `global'
collect2: ld returned 1 exit status
```

编译器在链接阶段有一步重要的动作叫符号解析与重定位。链接器会将所有中间文件的数据，代码，符号分别合并到一起，并计算出链接后的虚拟基地址。比如".text"段从 0x1000开始，".data"段从0x2000开始。接着链接器会根据基址计算各个符号(global)的相对虚拟地址。

当编译器发现在符号表中找不到global的地址时就会报出 `undefined reference to `global`.`说明在静态链接的过程中编译器必须在编译链接阶段完成对所有符号的链接。

如果使用PIE方式将extern_var.c编译成一个share library会出现什么情况呢？

```
gcc -o extern_var.so extern_var.c -shared -fPIC
```

程序能够顺利编译通过生成extern_var.so。但运行时会报错，因为装载时找不到global符号目标地址。这说明-fPIC选项生成了地址无关代码。将静态链接时没有找到的glob

那么在编译链接阶段，链接器是如何将这个缺失的目标地址在代码段中进行地址引用的呢？

链接器巧妙的用一张中间表GOT(Global Offset Table)来解决被引用符号缺失目标地址的问题。如果在链接阶段(jing
tai)发现一个不能确定目标地址的符号。链接器会将该符号加到GOT表中，并将所有引用该符号的地方用该符号在GOT表中的地址替换。到装载阶段动态链接器会将GOT表中

当程序执行到符号对应的代码时，程序会先查GOT表中对应符号的位置，然后根据位置找到符号的实际的目标地址。

地址无关代码的生成方式

所谓地址无关代码要求程序被加载到内存中的任意地址都能够正常执行。所以程序中对变量或函数的引用必须是相对的，不能包含绝对地址。

比如如下伪汇编代码：

PIE方式：代码可以运行在地址100或1000的地方

```
100: COMPARE REG1, REG2
101: JUMP_IF_EQUAL CURRENT+10
...
111: NOP
```

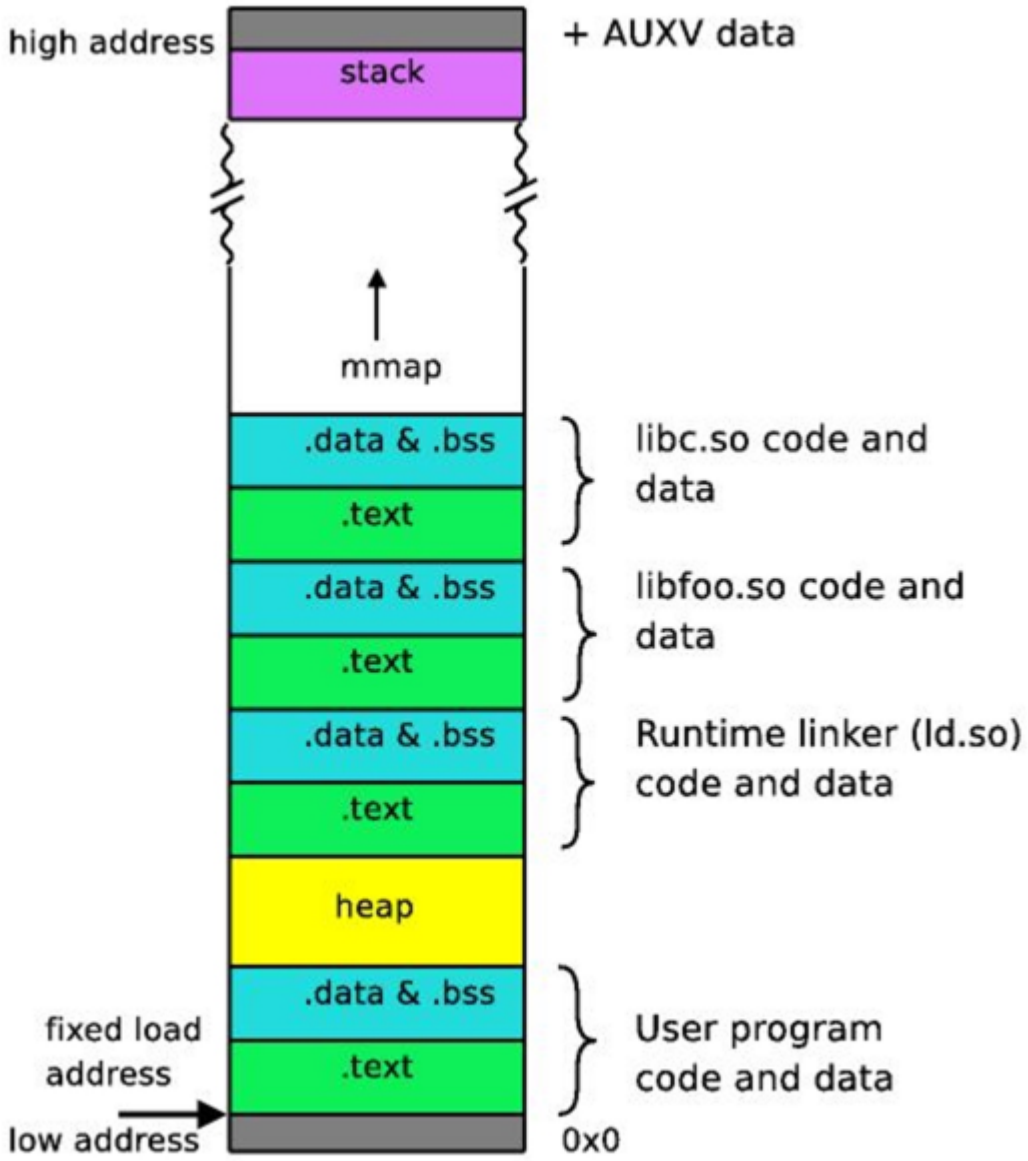
Non-PIE: 代码只能运行在地址100的地方

```
100: COMPARE REG1, REG2
101: JUMP_IF_EQUAL 111
...
111: NOP
```

因为可执行程序的代码段只有读和执行属性没有写属性，而数据段具有读写属性。要实现地址无关代码，就要将代码段中须要改变的绝对值分离到数据段中。在程序加载时可

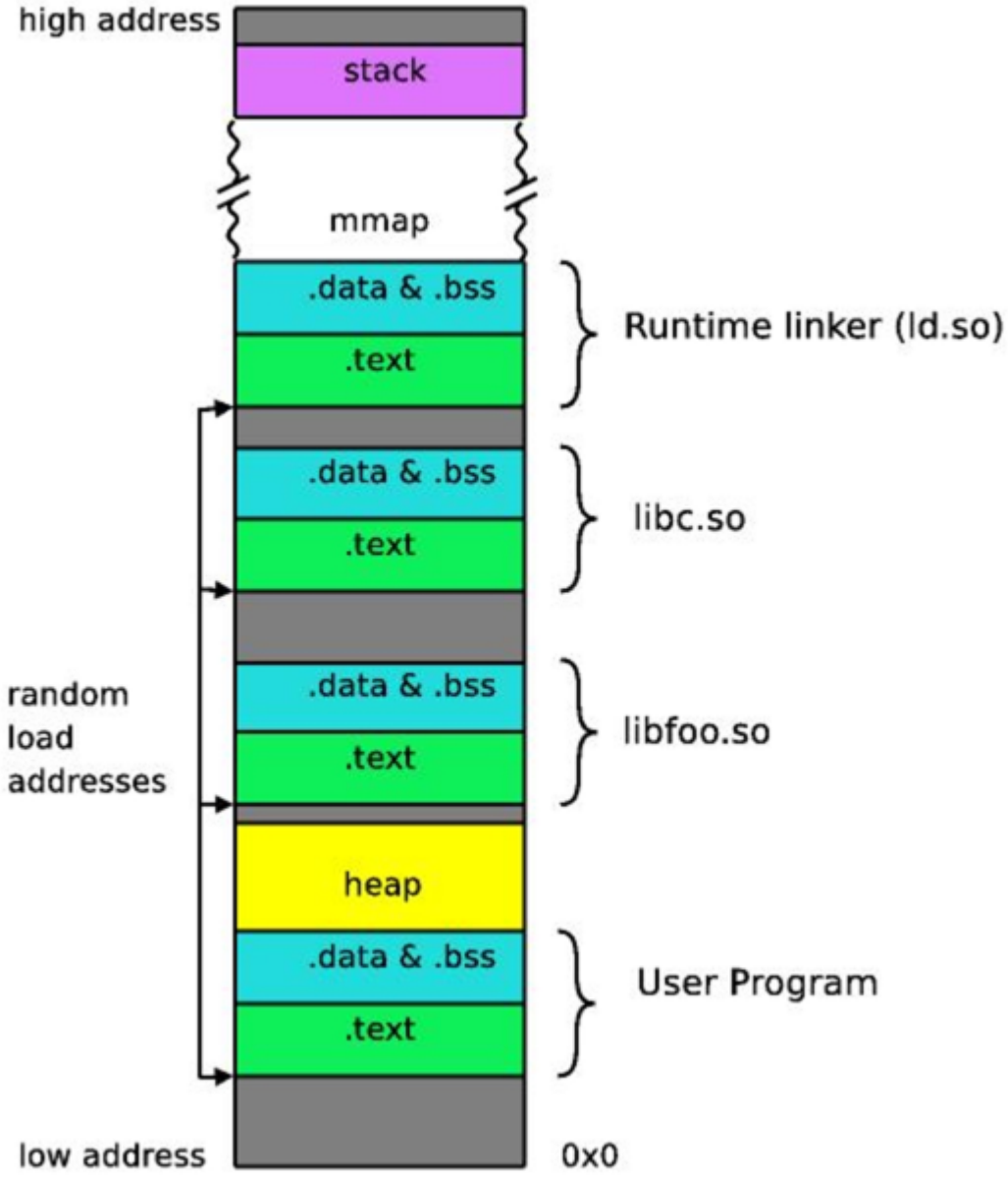
PIE和Non-PIE程序在内存中映射方式

在Non-PIE时程序每次加载到内存中的位置都是一样的。



执行程序会在固定的地址开始加载。系统的动态链接器库ld.so会首先加载，接着ld.so会通过.dynamic段中类型为DT_NEED的字段查找其他需要加载的共享库。并依次将它

而对于通过PIE方式生成的执行程序，因为没有绝对地址引用所以每次加载的地址也不尽相同。



不仅动态链接库的加载地址不固定，就连执行程序每次加载的地址也不一样。这就要求ld.so首先被加载后它不仅要负责重定位其他的共享库，同时还要对可执行文件重定位。

PIE与编译器选项

GCC编译器用于生成地址无关代码的参数主要有-fPIC, -fPIE, -pie。

其中-fPIC,
-fPIE属于编译时选项，分别用于生成共享库和可执行文件。它们能够使编译阶段生成的中间代码具有地址无关代码的特性。但这并不代表最后生成的可执行文件是PIE的。还

一个标准的PIE程序编译设置如下：

```
gcc -o sample_pie sample.c -fPIE -pie
```

在gcc中使用编译选项与是否生成PIE可执行文件对应关系如下：

-fPIE	-fPIC	-pie	ELF TYPE
Y	N	N	EXEC
N	Y	N	EXEC
Y	N	Y	DYN
N	Y	Y	DYN

Type为DYN的程序支持PIE，EXEC类型不支持。DYN, EXEC与PIE对应关系详见后文。

ASLR在Android中的应用

PIE属于ASLR的一部分，如上节提到ASLR包括对Stack, Heap, Libs and mmap, Executable, Linker, VDSO的随机化。

而支持PIE只表示对Executable实现了ASLR。随着Android的发展对ASLR的支持也逐渐增强。

ASLR in Android 2.x

Android对ASLR的支持是从Android 2.x开始的。2.x只支持对Stack的随机化。

ASLR in Android 4.0

而在4.0，即其所谓的支持ASLR的版本上，其实ASLR也仅仅增加了对libc等一些shared libraries进行了随机化，而对于heap, executable和linker还是static的。

对于heap的随机化来说，可以通过

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

来开启。

而对于executable的随机化，由于大部分的binary没有加GCC的-pie -fPIE选项，所以编译出来的是EXEC，而不是DYN这种shared object file，因此不是PIE（Position Independent Executable），所以没有办法随机化；

同样的linker也没有做到ASLR。

ASLR in Android 4.1

终于，在4.1 Jelly Bean中，Android支持了所有内存的ASLR。在Android 4.1中，基本上所有binary都被编译和连接成了PIE模式（可以通过readelf查看其Type）。所以，相比于4.0，4.1对Heap，executable和linker都提供了ASLR的支持。

ASLR in Android 5.0

5.0中Android抛弃了对non-PIE的支持，所有的进程均是ASLR的。如果程序没有开启PIE，在运行时会报错并强制退出。

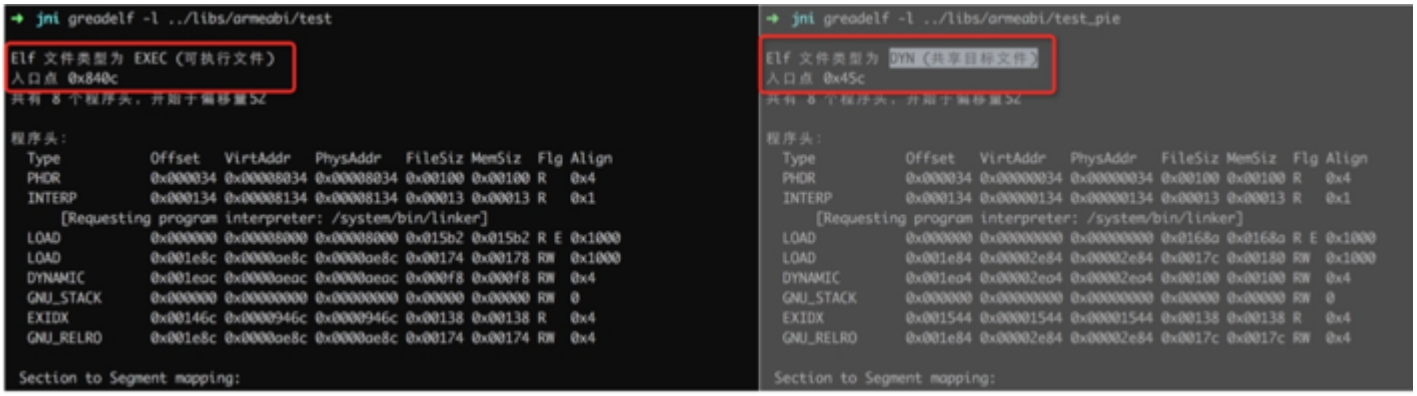
PIE程序运行在Android各版本

Type	2.x	4.0	4.1	5.0
PIE	N	N	Y	Y
Non-PIE	Y	Y	Y	N

支持PIE的可执行程序只能运行在4.1+的版本上。 在4.1版本之前运行会出现crash。而Non-PIE的程序，在5.0之前的版本能正常运行，但在5.0上会crash。

如何检测是否开启PIE

未开启PIE的执行程序用readelf查看其文件类型应显示EXEC(可执行文件)，开启PIE的可执行程序的文件类型为DYN(共享目标文件)。另外代码段的虚拟地址总是从0开始。



为什么检测DYN就可以判断是否支持PIE？

DYN指的是这个文件的类型，即共享目标文件。那么所有的共享目标文件一定是开启了PIE的吗？我们可以从源码中寻找答案。查看glibc/glibc-2.16.0/elf/dl-load.c中的代码

```
struct link_map *
_dl_map_object_from_fd (const char *name, int fd, struct filebuf *
      char *realname, struct link_map *loader, int l_type,
      int mode, void **stack_endp, Lmid_t nsid)
{
...
    if (__builtin_expect (type, ET_DYN) == ET_DYN)
      {
  /* This is a position-independent shared object.  We can let the
     kernel map it anywhere it likes, but we must have space for a
     the segments in their specified positions relative to the fir
     So we map the first segment without MAP_FIXED, but with its
     extent increased to cover all the segments.  Then we remove
     access from excess portion, and there is known sufficient spa
     there to remap from the later segments.

     As a refinement, sometimes we have an address that we would
     prefer to map such objects at; but this is only a preference,
     the OS can do whatever it likes. */
  ElfW(Addr) mappref;
  mappref = (ELF_PREFERRED_ADDRESS (loader, maplength,
           c->mapstart & GLRO(dl_use_load_bias))
       - MAP_BASE_ADDR (l));

  /* Remember which part of the address space this object uses.  *
  l->l_map_start = (ElfW(Addr)) __mmap ((void *) mappref, maplengt
                c->prot,
                MAP_COPY|MAP_FILE,
                fd, c->mapoff);
...
      }

    /* This object is loaded at a fixed address.  This must never
       happen for objects loaded with dlopen().  */
    if (__builtin_expect ((mode & __RTLD_OPENEXEC) == 0, 0))
      {
  errstring = N_("cannot dynamically load executable");
  goto call_lose;
      }

    /* Notify ELF_PREFERRED_ADDRESS that we have to load this one
       fixed.  */
    ELF_FIXED_ADDRESS (loader, c->mapstart);


    /* Remember which part of the address space this object uses.
    l->l_map_start = c->mapstart + l->l_addr;
    l->l_map_end = l->l_map_start + maplength;
    l->l_contiguous = !has_holes;

    while (c < &loadcmds[nloadcmds])
      {
  if (c->mapend > c->mapstart
      /* Map the segment contents from the file.  */
      && (__mmap ((void *) (l->l_addr + c->mapstart),
      c->mapend - c->mapstart, c->prot,
      MAP_FIXED|MAP_COPY|MAP_FILE,
      fd, c->mapoff)
    == MAP_FAILED))
```

从源码可知，如果加载类型不为ET_DYN时调用mmap加载文件时会传入MAP_FIXED标志。将程序映射到固定地址。

阿里聚安全开发中建议

1. 针对Android 2.x-4.1之前的系统在编译时不要使用生成PIE的选项。
2. 在Android 4.1以后的版本必须使用PIE生成Native程序，提高攻击者中的攻击成本。

3. 在版本上线前使用阿里聚安全漏洞扫描系统进行安全扫描，将安全隐患阻挡在发布之前。

Reference

- https://en.wikipedia.org/wiki/Position-independent_code
- http://www.openbsd.org/papers/nycbsdcon08-pie/
- https://source.android.com/security/enhancements/enhancements50.html
- http://ytliu.info/blog/2012/12/09/aslr-in-android/
- https://codywu2010.wordpress.com/2014/11/29/about-elf-pie-pic-and-else/
- http://www.cnblogs.com/huxiao-tee/p/4660352.html
- http://stackoverflow.com/questions/5311515/gcc-fpic-option

作者：呆狐@阿里聚安全，更多Android、iOS技术文章，请访问阿里聚安全博客

上一篇：你出要求我写工具 下一篇：Java Web 漏洞生態食物鏈

1. 1 条回复

笑然 2016-12-08 09:28:09

想看潘老师的书，书的名字吸引了我 哈哈哈

0 回复Ta
