**Rootcheck源码分析**

作者：李强 日期：2017年8月

本代码分析基于pop5-6 4g项目代码，以rootcheck代码为主线分析。先给出一张流程图（有点简陋哈）：



rootcheck功能的代码位置：

pop564g/vendor/jrdcom/system/root\_check/ root\_check.h

pop564g/vendor/jrdcom/system/root\_check/ root\_check.c

# 1.检测标识码

从main()函数开始分析，按照源代码的顺序进行分析，第一模块代码如下。

root\_check.c：

**if** **((**fd **=** open**(**TRACEPATITION**,** O\_RDONLY**))** **>** 0**)** **{**

lseek**(**fd**,** FLAGOFFSET**,** SEEK\_SET**);**

read**(**fd**,** buf**,** ROOTFLAGLEN**);**

close**(**fd**);**

**if** **(!**strncmp**(**ROOTFLAG**,** buf**,** ROOTFLAGLEN**))** **{**

property\_set**(**"persist.su\_flag"**,** "1"**);**

exit**(**1**);**

**}**

**}**

TRACEPATITION、ROOTFLAG、ROOTFLAGLEN、FLAGOFFSET在root\_check.h有如下定义：

#define ROOTFLAG "109f10eed3f021e3"

#define ROOTFLAGLEN strlen(ROOTFLAG)

#define FLAGOFFSET (2\*1024\*1024)

#define TRACEPATITION "/dev/block/platform/mtk-msdc.0/11230000.msdc0/by-name/proinfo

代码的功能就是取出proinfo中偏移了FLAGOFFSET（2M）的位置处和ROOTFLAG相同长度（16B）的标识码与ROOTFLAG标识码（109f10eed3f021e3）进行对比（问题：为什么这个image的标识码是109f10eed3f021e3）。如果相同，就调用property\_set函数将persist.su\_flag设置为1。

ROOTFLAG是用来表示是否被root的标识码，proinfo是一个分区，考虑到proinfo分区的信息敏感，所以对偏移了2M进行记录此信息。即使手机root后又回退root，也会将相关的信息写入到proinfo中，除非是擦除了proinfo分区。所以，进行rootcheck第一步是进行proinfo检测。

# 2.检测su文件

那么，proinfo的表示码是什么时候写入的呢？是之前进行root检测的时候写入的，方便下次的查看，下面就分析满足那些条件要写入这个被root的标识码。

root\_check.c：

**else** **if(**check\_su\_exists**())** **{**

setflag**(**"su exists!"**);**

exit**(**1**);**

**}**

当检测完proinfo发现现实没有记录被root，但是这并不代表手机没有被root，而是说只是没有被root的记录而已。对su文件进行检测，源代码如下：

root\_check.c：

static int check\_su\_exists**()**

**{**

/\*concat pathes, access them\*/

size\_t sz\_files **=** **sizeof(**g\_breaker\_filename**)/sizeof(**char **\*);**

size\_t sz\_folders **=** **sizeof(**monitored\_folders**)/sizeof(**char **\*);**

int i**,**j**;**

char **\***path **=** **(**char **\*)**malloc**(sizeof(**char**)** **\*** MAX\_PATH\_LEN**);**

int fid**;**

**for(**i **=** 0**;** i **<** sz\_files**;** **++**i**){**

**for(**j **=** 0**;** j **<** sz\_folders**;** **++**j**){**

memset**(**path**,** 0**,** MAX\_PATH\_LEN**);**

strncpy**(**path**,** monitored\_folders**[**j**],** strlen**(**monitored\_folders**[**j**]));**

strncpy**(**path **+** strlen**(**monitored\_folders**[**j**]),** "/"**,** 1**);**

strncpy**(**path **+** strlen**(**monitored\_folders**[**j**])** **+** 1**,** g\_breaker\_filename**[**i**],**

strlen**(**g\_breaker\_filename**[**i**])+**1**);**

#ifdef DEBUG

INFO**(**"[%s]:%d open file %s\n"**,** MOD**,** \_\_LINE\_\_**,** path**);**

#else

fid **=** open**(**path**,** O\_RDONLY**);**

**if(**fid **>** 0**){**

close**(**fid**);**

**return** 1**;**

**}**

#endif

**}**

**}**

**return** 0**;**

**}**

代码就是对su文件可能出现的三个系统指令目录：

root\_check.h：

static const char **\*** monitored\_folders**[]** **=** **{**

"/sbin"**,**

"/system/bin"**,**

"/system/xbin"

**};**

排查看是否有名字为以下列表中的文件，判断的方法就是利用open函数去打开拼接的文件地址，如果能打开就return 1；

root\_check.h：

static const char **\***g\_breaker\_filename**[]** **=** **{**

"su"**,**

"Su"**,**

"sU"**,**

"SU"

**};**

Return 1的结果就是在root\_check.c的main函数中药执行

setflag**(**"su exists!"**);**

这个setflag代码就是来设置proinfo分区里面的rootflag标识码的。源码如下：

root\_check.c：

static void setflag**(**const char **\***msg**)**

**{**

int fd **=** **-**1**;**

close**(**open**(**"/tctpersist/root\_flag"**,** O\_WRONLY **|** O\_CREAT**,** 0400**));**

property\_set**(**"persist.su\_flag"**,** "1"**);**

sync**();**

fd **=** open**(**TRACEPATITION**,** O\_RDWR**|**O\_SYNC**);**

**if** **(**fd **<** 0**)** **{**

ERROR**(**"[%s]:%d open %s failed: %s\n"**,** MOD**,** \_\_LINE\_\_**,** TRACEPATITION**,** strerror**(**errno**));**

**return;**

**}**

lseek**(**fd**,** FLAGOFFSET**,** SEEK\_SET**);**

write**(**fd**,** ROOTFLAG**,** ROOTFLAGLEN**);**

**if** **(**msg **!=** **NULL)** **{**

lseek**(**fd**,** FLAGOFFSET **+** ROOTFLAGLEN **+** ROOTFLAGLEN**,** SEEK\_SET**);**

write**(**fd**,** msg**,** strlen**(**msg**)** **+** 1**);**

**}**

fsync**(**fd**);**

close**(**fd**);**

**}**

/tctpersist/root\_flag这块没懂？

先是设置property\_set**(**"persist.su\_flag"**,** "1"**)**（此处不管是检测到什么都要设置该flag是为了方便查看），然后在TRACEPATITION（也就是proinfo）写入ROOTFLAG标识码，并且将传来的msg写在后面。当检测到su文件的时候，就认为被root（因为user版本是不带su文件的），这就是之前的第一步检测的proinfo文件，一旦检测里面有被root的标识码就直接告知系统被root，后续的检测不用进行了，其实很好理解，后续的检测就是为了在proinfo中写入手机被root的标识码。

那么root手机为什么要把su文件放入到系统指令目录下？为了获取root权限，需要使用到su文件进行提权，使用su命令将权限提升到root级别。一般需要执行以下3步：

adb push su **/**system**/**bin

adb push SuperUser.apk **/**system**/**app

adb shell chmod 4755 **/**system**/**bin**/**su

由于以下或者其他原因：

1. user版本的/system 是只读的；
2. chmod命令是需要root权限去执行的；
3. 手机中的su的权限会被设置为755，或者直接删除su文件。

为了执行以上3步，不同的破解软件使用的不同，但是最终都是讲su文件放到了对应的目录中。对于检测软件只用关心检测是否存在su文件即可。(获取root权限的su文件是被修改的su文件，跳过了对用户是否是root用户或者shell用户的检测，具体可以参考别的资料。这样所有进程都可以拥有root权限，所以一般root后的手机需要安装类似superuser.apk对root权限进行管理。)

# 3.adb

root\_check.c：

/\* Check ro.secure & ro.debuggable \*/

property\_get**(**"ro.secure"**,** value**,** ""**);**

**if** **(**strcmp**(**value**,** "1"**))** **{**

INFO**(**"[%s]:%d eng build\n"**,** MOD**,** \_\_LINE\_\_**);**

setflag**(**"ro.secure"**);**

exit**(**1**);**

**}**

property\_get**(**"ro.debuggable"**,** value**,** ""**);**

**if** **(**strcmp**(**value**,** "0"**))** **{**

INFO**(**"[%s]:%d debug build\n"**,** MOD**,** \_\_LINE\_\_**);**

setflag**(**"ro.debuggable"**);**

exit**(**1**);**

**}**

这个部分的检测代码很好理解，就是检测ro.secure的值是否为1，如果不是，执行setflag函数（该函数之前介绍了，原理一样，就是最后加的msg不一样）；检测ro.debuggable的值是否为0，如果不是，执行setflag函数。这两个系统属性值和root有什么关系？要考虑到root的目的，root是为了修改/system分区的内容，adb shell默认进入手机不是root权限的，没有root权限是不能修改/system分区的文件的。为了使adb shell拥有root权限，可以修改根目录下的default.prop提权，修改的文件就是ro.secure和ro.debuggable文件。为什么是这两个属性就要看adbd的源码，一般的adb源码在：

android/system/core/adb/adb.c

在pop564g项目中是在

pop546g/system/core/adb/daemon/main.cpp（具体位置可以在Android.mk文件中查看）

adbd在开机的时候是root，后来被降级了，降级函数就是在这个main.cpp中的函数的should\_drop\_privileges()，源码如下：

main.cpp：

static bool should\_drop\_privileges**()** **{**

#ifdef MTK\_ALLOW\_ADBD\_ROOT//系统允许root才行，如果不允许直接退出

**return** **false;**

#endif

#if defined(ALLOW\_ADBD\_ROOT)

char value**[**PROPERTY\_VALUE\_MAX**];**

// The properties that affect `adb root` and `adb unroot` are ro.secure and

// ro.debuggable. In this context the names don't make the expected behavior

// particularly obvious.

//

// ro.debuggable:

// Allowed to become root, but not necessarily the default. Set to 1 on

// eng and userdebug builds.

//

// ro.secure:

// Drop privileges by default. Set to 1 on userdebug and user builds.

property\_get**(**"ro.secure"**,** value**,** "1"**);**

bool ro\_secure **=** **(**strcmp**(**value**,** "1"**)** **==** 0**);**

property\_get**(**"ro.debuggable"**,** value**,** ""**);**

bool ro\_debuggable **=** **(**strcmp**(**value**,** "1"**)** **==** 0**);**

// Drop privileges if ro.secure is set...

bool drop **=** ro\_secure**;**

property\_get**(**"service.adb.root"**,** value**,** ""**);**

bool adb\_root **=** **(**strcmp**(**value**,** "1"**)** **==** 0**);**

bool adb\_unroot **=** **(**strcmp**(**value**,** "0"**)** **==** 0**);**

// ... except "adb root" lets you keep privileges in a debuggable build.

**if** **(**ro\_debuggable **&&** adb\_root**)** **{**

drop **=** **false;**

**}**

// ... and "adb unroot" lets you explicitly drop privileges.

**if** **(**adb\_unroot**)** **{**

drop **=** **true;**

**}**

**return** drop**;**

#else

**return** **true;** // "adb root" not allowed, always drop privileges.

#endif // ALLOW\_ADBD\_ROOT

**}**

方法的注释中很清楚的说明了为什么是ro.debuggable和ro.secure两个系统属性。这两个参数会影响到adb root和adb unroot。为了让adb获取root权限，就是让adb的权限不被降级，需要修改这两个参数，所以检测这两个参数是否被修改，可以用来作为手机是否被root的标准。

# 4.dm-verity

root\_check.c：

//check system dm-verity status

**if** **(**check\_system\_mount**(**"/proc/mounts"**))** **{**

setflag**(**"dm-verity"**);**

exit**(**1**);**

**}**

分析check\_system\_mount方法（代码比较长，去掉不关键的代码） ：

static int check\_system\_mount**(**const char **\***file\_path**)**

**{**

…………………

**while** **((**len **=** getline**(&**line**,** **&**alloc\_len**,** mount\_file**))** **!=** **-**1**)** **{**

…………………

//printf("mount\_point = %s\n", mount\_point);

**if(**0 **==** strcmp**(**SYSTEM\_MOUNT\_POINT**,** mount\_point**))** **{**

INFO**(**"[%s]:%d bld\_device = %s\n"**,** MOD**,** \_\_LINE\_\_**,** bld\_device**);**

**if(**0 **==** strcmp**(**SYSTEM\_BLK\_DEVICE**,** bld\_device**))** **{**

printf**(**"[%s]:%d dm-verity is enabled!\n"**,** MOD**,** \_\_LINE\_\_**);**

**}** **else** **{**

printf**(**"[%s]:%d dm-verity was disabled!!\n"**,** MOD**,** \_\_LINE\_\_**);**

mount\_status **=** 1**;**

**}**

**break;**

**}**

**}**

err**:**

fclose**(**mount\_file**);**

free**(**line**);**

**return** mount\_status**;**

**}**

SYSTEM\_MOUNT\_POINT、SYSTEM\_BLK\_DEVICE定义：

root\_check.h：

#define SYSTEM\_MOUNT\_POINT "/system"

#define SYSTEM\_BLK\_DEVICE "/dev/block/dm-0"

代码的功能实际上就是检测/dev/block/dm-0是否挂载在/system挂载点。如果不是判定手机被进行过root操作。dm-verify功能，可以检测系统是否被篡改，以此保存系统的完整性，/system分区就是被dm-verity保护的分区之一。如果dm-verity正常开启，那么挂载设备会是dm-0这种命名（具体为什么这样不清楚！！！），所以可以通过该检测来判断dm-verity是否正常工作。有dm-verity的保护，即使/system被修改，手机重启后，修改也会被撤销（不知道准确不？）。

# 5.SELinux

root\_check.c：

//check selinux status

**if** **(!**check\_selinux\_status**(**"/sys/fs/selinux/enforce"**))** **{**

setflag**(**"selinux"**);**

exit**(**1**);**

**}**

在手机中，/sys/fs/selinux/enforce记录着SELinux是否是处于enforcing（强制模式），如果是enforce保存1，没有则为0。check\_selinux\_status方法就是检测该值是否为0，如果是0则判定进行过root操作。Root与SELinux的开启有什么关系？在现在的Linux系统中不是用了root权限就是想干什么就干什么，还有一个SELinux限制，也就是即使有了root权限也无法修改系统权限。SELinux设计了一个新的安全模型，叫MAC（Mandatory Access Control），翻译为**强制访问控制**。核心思想如下：即任何进程想在SELinux系统中干任何事情，都必须先在安全策略配置文件中赋予权限。凡是没有出现在安全策略配置文件中的权限，进程就没有该权限。在SELinux眼中没有root的概念。所以要获取想干什么就干什么的权限是需要关闭SELinux的，所以通过检测SELinux是否被关闭，这样可以用来判断手机是否有root的操作。

# 6. monitor

使用do\_monitor()函数去监听su文件被进行的操作。通知相关的信息，可以实时的看到。并且发现有su文件就去在proinfo文件中记录，这样的好处是，在手机root的时候就记录下被root的操作，防止手机被root后将该函数删除，这样就无法再proinfo中记录被root的操作了。