Android系统
BOOT分区病毒检测与清除

网秦
联系方式:yuandengkai@nq.com
330322177@qq.com

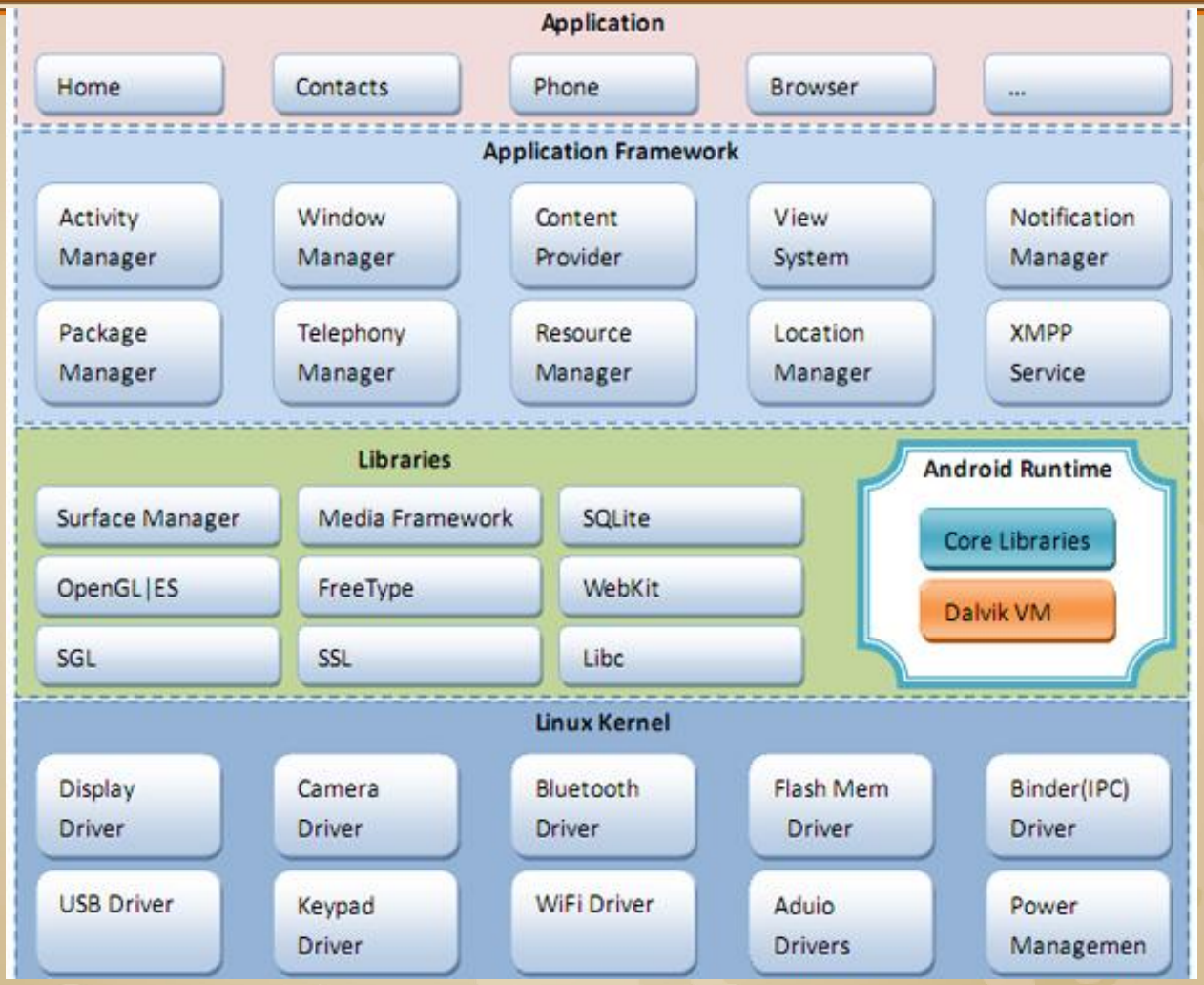
Apr 16,17 2014 Shanghai

{xKungfoo 2014}

主要内容

- Android系统各分区简介
- 主流手机存储设备类型
- Boot分区病毒简介
- 典型BOOT分区病毒例子分析
- BOOT分区病毒的检测
- BOOT分区病毒的清除
- 演示

Android体系架构



Android版本内核历史

Android版本	Linux内核对应版本
1.5(Cupcake)	2.6.27
1.6(Donut)	2.6.29
2.0/2.0.1/2.1(Eclair)	2.6.29
2.2/2.2.1(Froyo)	2.6.32
2.3(Gingerbread)	2.6.35
3.0.1/3.1/3.2(Honeycomb)	2.6.36
4.0(Ice Cream Sandwich)	3.0.1
4.1/4.2/4.3(Jelly Bean)	3.4.0
4.4(KitKat)	3.4.0

Android加电启动过程1

阶段	步骤	备注
BootLoader		bootable\bootloader\legacy\usbloader
	init.S	初始化栈，将BSS段置零，调用main.c中的_main()函数
	main.c	显示”USB FastBoot”,从flash中启动，或者在usb_poll()函数中循环等待PC的连接
Linux kernel		设置系统，加载驱动，并且运行第一个进程init
The *init* process	设置文件系统	创建并挂载目录(/dev,/proc,/sys)
	执行init.rc脚本	Android特有语法的启动脚本

Android加电启动过程2

	设置控制	
	显示”ANDROID”字符串	文本消息写入到/dev/tty0设备
	Zygote	Init.rc中配置的Zygote进程启动dalvik虚拟机并且启动system server进程
	开机动画	在启动过程中显示开机动画
Framework	……	…….

Linux启动简介

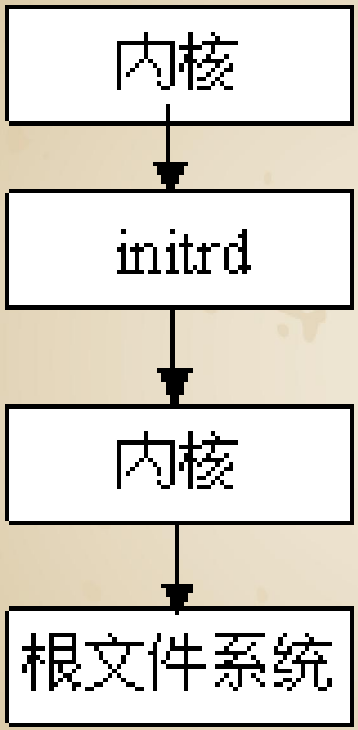
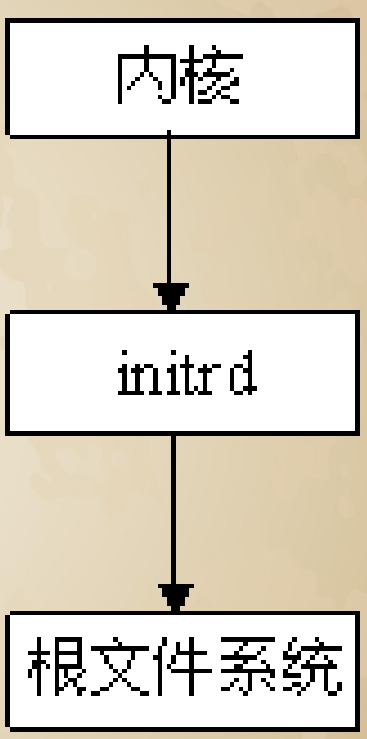


image-initrd 的处理阶段



cpio-initrd 的处理阶段

Android系统各分区简介

- hboot分区
- boot分区
- radio分区
- recover分区
- system分区
- userdata分区
- cache分区
- sd-ext分区

主流手机存储设备类型

- /dev/block/mmcblk设备
- /dev/block/loop设备
- /dev/block/ram设备
- /dev/emmc设备
- /dev/mtd设备

Boot分区病毒简介

- Boot分区病毒是所有在开机时比系统内核更早加载，实现内核劫持的技术的一类病毒.
- Boot分区病毒可以含有一个病毒体独立运行,也可以含有多个病毒体协同工作.
- Boot分区病毒除了一般病毒的破坏力外,还往往因为其较高的操作权限,能对系统进行较大的破坏.

Boot分区病毒难以清除的原因

- 1 Boot分区是一个虚拟的内存文件系统，普通的直接对其分区内文件的操作不会被持久化

- 2 病毒启动时机相对来说比较早，可以利用时间差做一些自我防御性的工作

例:OldBoot病毒的特征

- 一 藏身于BOOT分区
- 二 多病毒体(ELF/SO/APK)

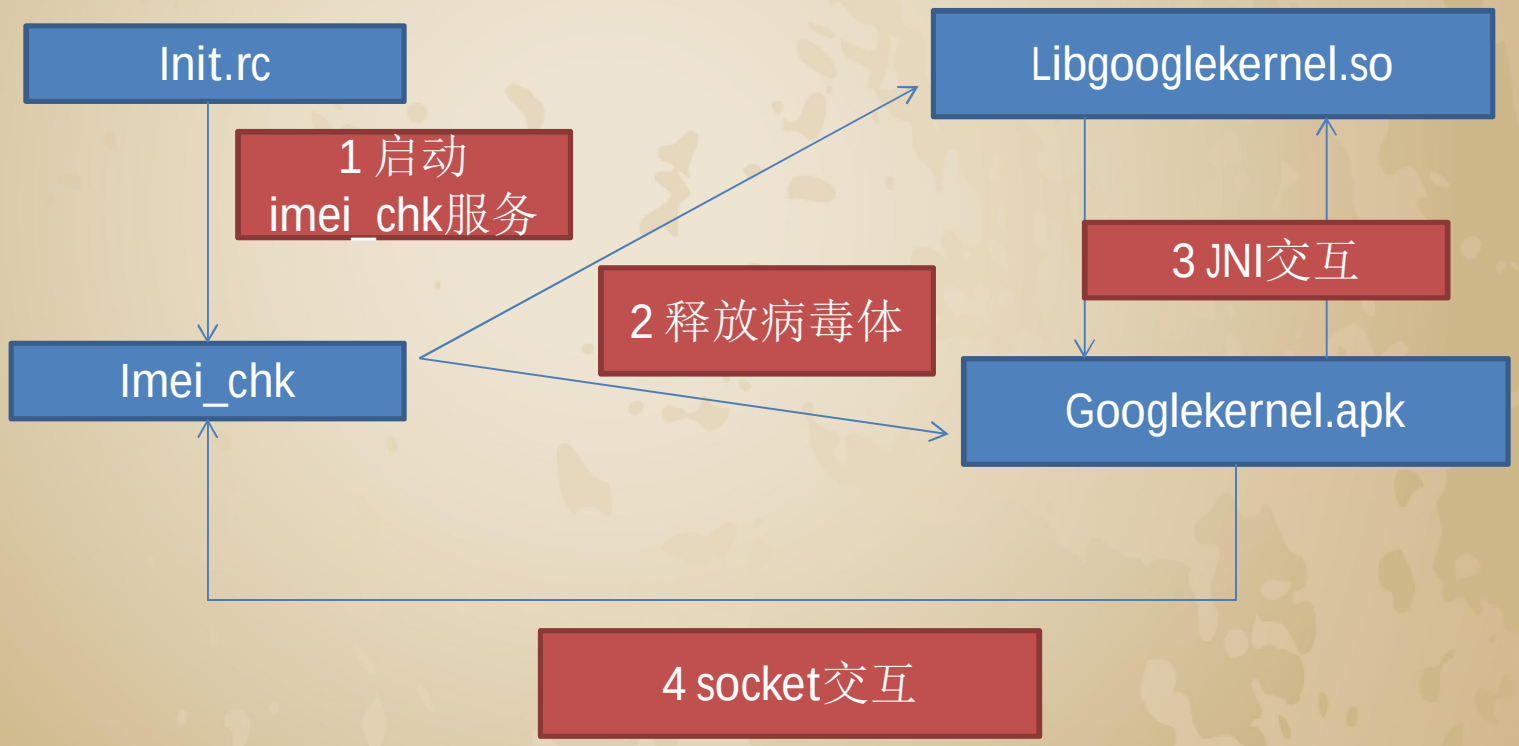
1 imei_chk(ELF)藏身于/sbin下，init.rc中配置有imei_chk的服务启动信息

2 googlekernel.apk位于/system/app下

3 Libgooglekernel.so位于/system/lib目录下

4 Imei_chk作为系统服务优先启动，释放apk和so的病毒体(前提是这2个病毒体不存在)

Oldboot病毒调用关系



BOOT分区病毒的检测

- 1 首先必须要有对敏感系统目录操作的权限（一般须具有ROOT权限）
- 2 对关键目录中的文件进行扫描
  - 采用特征串匹配的方法
  - 采用HASH比对的方法
- 3 对Oldboot的实体检测就是对boot.img文件内ramdisk的检测

Img文件实际布局

Header
Header Padding
Kernel
Kernel padding
Ramdisk.gz
Ramdisk padding
Second
Second padding

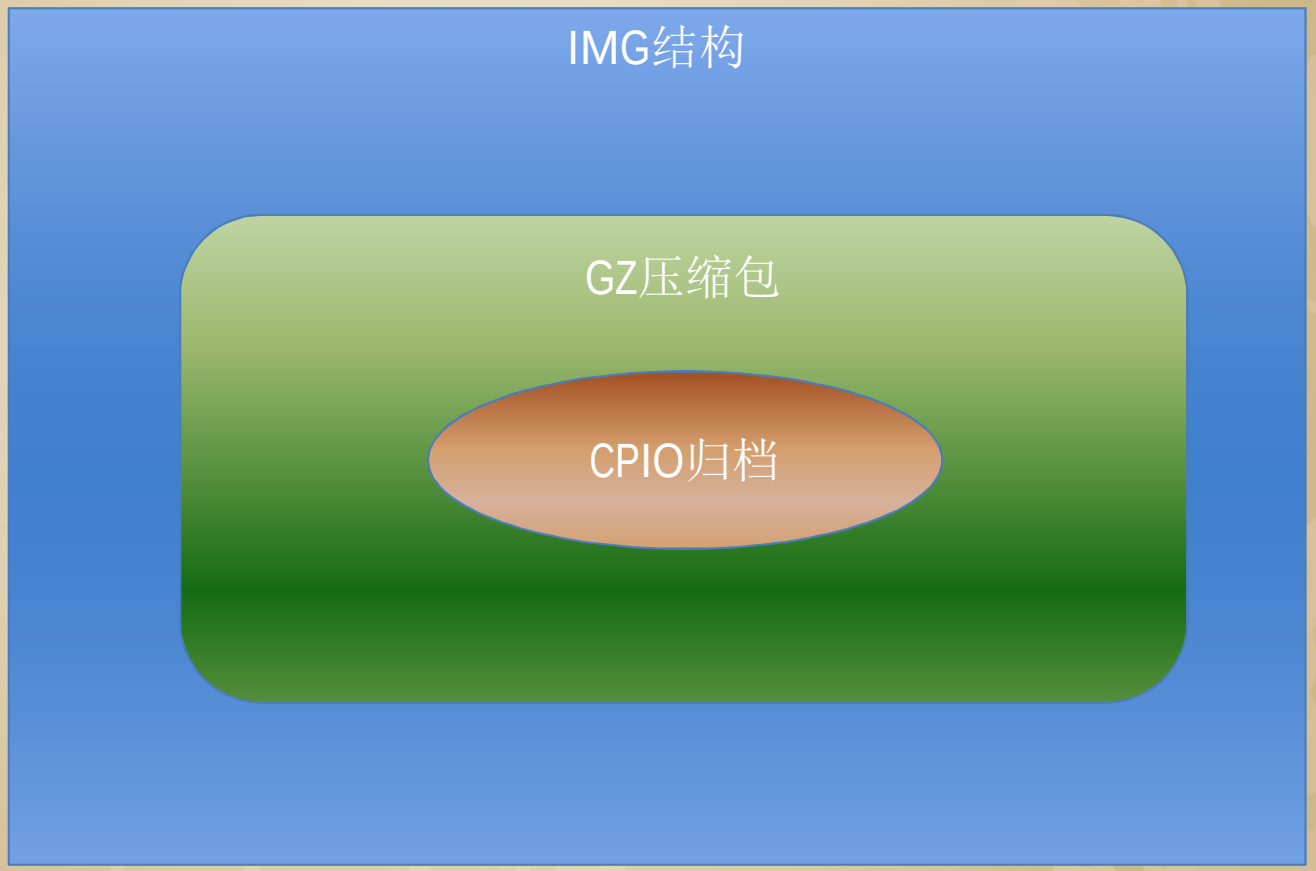
上图为boot.img的文件布局结构，数据按照header里page_size的数值进行对齐，不足部分使用全0的padding补齐

IMG头部结构体定义(c/cpp)

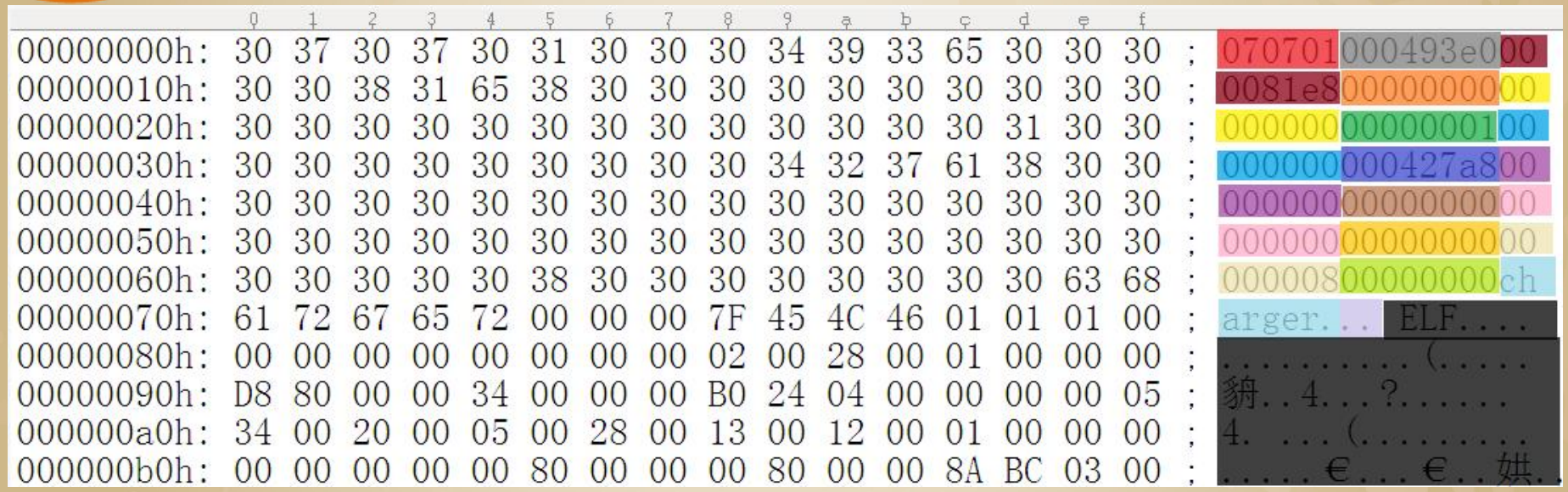
```
struct boot_img_hdr
{
    unsigned char magic[BOOT_MAGIC_SIZE];

    unsigned kernel_size;  /* size in bytes */
    unsigned kernel_addr;  /* physical load addr */

    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */

    unsigned second_size;  /* size in bytes */
    unsigned second_addr;  /* physical load addr */

    unsigned tags_addr;    /* physical addr for kernel tags */
    unsigned page_size;    /* flash page size we assume */
    unsigned unused[2];    /* future expansion: should be 0 */

    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */

    unsigned char cmdline[BOOT_ARGS_SIZE];

    unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};
```

上图为boot.img的头部结构定义

CPIO位置



CPIO格式(HEX例子)



```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 30 37 30 37 30 31 30 30 30 34 39 33 65 30 30 30 ; 070701000493e000
00000010h: 30 30 38 31 65 38 30 30 30 30 30 30 30 30 30 30 ; 0081e8000000000
00000020h: 30 30 30 30 30 30 30 30 30 30 30 30 30 31 30 30 ; 000000000000100
00000030h: 30 30 30 30 30 30 30 30 30 34 32 37 61 38 30 30 ; 000000000427a800
00000040h: 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ; 000000000000000
00000050h: 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 ; 000000000000000
00000060h: 30 30 30 30 30 38 30 30 30 30 30 30 30 30 63 68 ; 000008000000ch
00000070h: 61 72 67 65 72 00 00 00 7F 45 4C 46 01 01 01 00 ; arger...ELF....
00000080h: 00 00 00 00 00 00 00 00 02 00 28 00 01 00 00 00 ; ..........(.....
00000090h: D8 80 00 00 34 00 00 00 B0 24 04 00 00 00 00 05 ; 貂..4...?......
000000a0h: 34 00 20 00 05 00 28 00 13 00 12 00 01 00 00 00 ; 4. ...(.........
000000b0h: 00 00 00 00 00 80 00 00 00 80 00 00 8A BC 03 00 ; .....€...€..姒..
```

CPIO格式解析(例子对照)

格式化后数据	释义	字段长度（字节）	对应颜色	备注
070701	Cpio项标志头	6(%06x)		Newc格式标志头
000493e0	项序号	8(%08x)		以000493e0作为起始序号，以后每一项递加1
000081e8	stat.st_mode	8(%08x)		文件类型和权限信息
00000000	stat.st_uid	8(%08x)		实际全部填充0
00000000	stat.st_gid	8(%08x)		实际全部填充0
00000001	stat.st_nlink	8(%08x)		实际填充1
00000000	stat.st_mtime	8(%08x)		实际全部填充0
000427a8	实际数据大小	8(%08x)		
00000000	volmajor	8(%08x)		实际全部填充0
00000000	volminor	8(%08x)		实际全部填充0
00000000	devmajor	8(%08x)		实际全部填充0
00000000	devminor	8(%08x)		实际全部填充0
00000008	项名称长度	8(%08x)		以NUL结尾的字符串，包含NUL的长度
00000000		8(%08x)		实际全部填充0
“charger”	项名称	变长(%s)		文件/目录/Link的名称
0	项名称字符串结尾	1(%c)		实际填充NUL
0	Padding数据	变长		补齐4字节数据对齐
ELF....	内容数据	变长		
0	Padding数据			补齐4字节数据对齐

CPIO文件项结尾

项名称为 ” TRAILER!!! ”

项实际数据大小为0

总数据的大小按照256字节对齐

查找boot分区的几种方法

- MTD (Memory Technology Device)
  - /proc/mtd
- EMMC (Embedded MultiMedia Card)
  - /proc/emmc
- MMC (MultiMedia Card)
  - /proc/partitions
  - parted
  - /dev/block/platform/msm_sdcc.1/by-name
- 遍历/dev
  - 特征串Android!

BOOT分区病毒的清除(思路一)

- 1 申请获取ROOT权限
- 2 遍历/dev/block设备下的设备，根据前8个字节(“ANDROID!”)确定boot.img格式的文件
- 3 根据Boot.img头部结构，定位ramdisk的文件起始位置及大小
- 4 提取ramdisk部分至磁盘文件，命名为ramdisk.gz
- 5 gzip解压ramdisk
- 6 cpio提取ramdisk的归档文件

BOOT分区病毒的清除(思路一续)

- 7 做修复操作(修改init.rc 删除/sbin/imei_chk)
- 8 cpio重打包所有文件成newc格式的文件mod_ramdisk
- 9 gzip压缩mod_ramdisk文件为mod_ramdisk.gz
- 10 dd命令将mod_ramdisk回写至提取设备
- 11 dd命令修正boot.img头部信息中ramdisk的相关部分

BOOT分区病毒的清除(思路二)

- 前面的定位查找思路不变(思路一前5个步骤)
- 直接对GZ解开的CPIO进行操作，修改INIT.RC中的相关配置项，使用空格替代(0x20),删除IMEI_CHK文件项
- 打包成GZ文件并回写会设备的固定偏移
- 依据新打包的GZ文件，修复IMG头部中RAMDISKSIZE字段

相关代码(JAVA层启动SU shell)

```
public static String execmd( String cmd ){
    String strRet = "";
    Process proc = null;
    BufferedOutputStream bos = null;
    BufferedReader br = null;
    Log.e("nick","执行命令:" + cmd );
    try {
        proc = Runtime.getRuntime().exec("su");
        bos = new BufferedOutputStream(proc.getOutputStream());
        br = new BufferedReader(new InputStreamReader(proc.getInputStream()));

        //写入执行命令
        bos.write( ( cmd + " 2>&1\n").getBytes() );
        bos.flush();

        //执行退出命令
        bos.write( "exit\n".getBytes() );
        bos.flush();
        bos.close();

        //读取返回结果
        while( true ){
            String strTmp;
            strTmp = br.readLine();

            if( strTmp == null ){
                break;
            }
            else{
                strRet += strTmp;
                strRet += "\n";
            }
        }
```

相关代码(判定ROM含毒代码)

```
//执行shell脚本dd出init.rc的内容
SuUtils.execmd("dd if=" + mInitFilePath + " of=" + mWorkDir + "/init.rc" );

//赋予权限
SuUtils.execmd("chmod 777 " + mWorkDir+"/init.rc" );

//查找关键字符串并写入到相应的文件
File tmpInit = new File( mWorkDir + "/init.rc" );
if( !tmpInit.exists() ){
    //dd命令执行失败
    return -1;
}

try {
    InputStream in = new FileInputStream( tmpInit );
    long fileLen = tmpInit.length();
    //Log.e("nick", "init.rc文件大小:"+fileLen);
    byte b[]=new byte[(int) fileLen];     //创建合适文件大小的数组
    in.read(b);    //读取文件中的内容到b[]数组
    in.close();

    //查找init.rc中的关键配置项
    String strContent = new String( b );
    if ( -1 != strContent.indexOf( "service imei_chk /sbin/imei_chk" ) ){
        //找到了oldboot的服务配置项，
        return 0;
    }
    else{
        //没找到关键配置项，安全
        return 1;
    }
```

相关代码(查找MTD设备boot分区)

```
//查找MTD分区
private String findBootMtdPartition(){
    String strPartition = null;

    String strFinder = "\"boot\"";
    int finderIndex = -1;

    //执行cat命令
    String strRet = SuUtils.execmd("cat /proc/mtd" );
    String[] strLines = strRet.split("\n");

    for( int i = 0; i < strLines.length; i++){
        if( strLines[i].contains(strFinder) ){
            String devs[] = strLines[i].split(":");
            strPartition = "/dev/mtd/" + devs[0];
        }
    }

    return strPartition;
}
```

相关代码(查找MMC设备boot分区)

```
//查找MMC分区
private String findBootMmcPartition(){
    String strPartition = null;
    String strFinder = " boot -> ";
    int finderIndex = -1;

    //执行ls -l命令查找
    String strRet = SuUtils.execmd("ls -l /dev/block/platform/msm_sdcc.1/by-name/" );
    String[] strLines = strRet.split("\n");

    for( int i = 0; i < strLines.length; i++){
        finderIndex = strLines[i].indexOf(strFinder);
        if( -1 !=  finderIndex ){
            strPartition = strLines[i].substring( finderIndex + strFinder.length() );
        }
    }

    return strPartition;
}
```

相关代码(暴力搜索boot分区)

```
String strPartition = null;                //搜索到的boot分区路径

//执行ls命令查找所有block设备
String strRet = SuUtils.execmd("ls /dev/block" );
String[] strLines = strRet.split("\n");
List<String> devs = new ArrayList<String>();

//依次读取设备前8个字节，判断是否为img格式
for( int i = 0; i < strLines.length; i++ ){
    String strTarget = SuUtils.execmd("dd if=/dev/block" + "/" + strLines[i] + " ibs=1 count=8");
    if( strTarget.contains("ANDROID!") ){
        devs.add("/dev/block" + "/" + strLines[i]);
    }
}
//依次循环处理多个img boot.img/recover.img
long ramsize = 0;              //获取ramdisk.gz大小
int index = 0;                 //img的索引

for( int j = 0; j < devs.size(); j++){
    //从设备上分离出头部,头部大小608个字节
    SuUtils.execmd("dd if=" + devs.get(j) + " of=" + mWorkDir+"/" + String.valueOf(j) + ".header ibs=1 count=608");
    SuUtils.execmd("chmod 777 " + mWorkDir+"/" + String.valueOf(j) + ".header" );

    //读取头部信息失败
    imgTools img = new imgTools();
    if( !img.openFile( mWorkDir+"/" + String.valueOf(j) + ".header" ) ){
        img.closeFile();
        continue;
    }

    //ramsize小的那个img就是boot.img
    if( (ramsize == 0) || (ramsize > img.getRamdiskSize()) ){
        ramsize = img.getRamdiskSize();        //获取ramdisk.gz大小
        index = j;
    }
    img.closeFile();
}
strPartition = devs.get( index );
return strPartition;
```

相关代码(定位分区&dump头部)

```
//查找boot分区
String bootPartition = findBootPartition();
if( bootPartition == null ){
    return retCode;
}

//dump boot.img头部 存为bootimg.header
SuUtils.execmd("dd if=" + bootPartition + " of=" + mWorkDir+"/bootimg.header ibs=1 count=608");
SuUtils.execmd("chmod 777 " + mWorkDir+"/bootimg.header" );

//打开头部读取相应的数据
imgTools img = new imgTools();
if( !img.openFile( mWorkDir+"/bootimg.header" ) ){
    img.closeFile();
    Log.e("nick", "无法打开文件");
    return retCode;
}

//获取ramdisk的大小
ramsize = img.getRamdiskSize();

//获取pagesize
pagesize = img.getPageSize();

//获取kernelsize
kernelsize = img.getKernelSize();
```

相关代码(定位ramdisk.gz的偏移)

```
//获取boot分区中ramdisk.gz的偏移
long getRamdiskOffset( long kernelsize, long pagesize ){
    long ramoff = 0;                                    //ramdisk的偏移
    if( kernelsize % pagesize == 0 )
    {
        ramoff =  kernelsize + pagesize;                //kernelsize正好是pagesize的整数倍
    }
    else
    {
        ramoff = ( kernelsize / pagesize + 1 + 1 ) * pagesize;
    }

    return ramoff;
}
```

相关代码(提取gz文件并解压)

```
//从设备上分离出ramdisk
SuUtils.execmd("dd if=" + bootPartition + " of=" + mWorkDir + "/oldramdisk.gz skip="
                    + String.valueOf(ramoff)  + " ibs=1 count=" + String.valueOf(ramsize));

//解压gzip文件
try {
    SuUtils.execmd("chmod 777 " + mWorkDir+"/oldramdisk.gz" );
    gzipUtils.decompress( mWorkDir+"/oldramdisk.gz", false );
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

相关代码(修复ramdisk文件)

```
//调用native函数修复
ramdiskTools fixer = new ramdiskTools();
boolean bInit = fixer.beginParse(mWorkDir+"/oldramdisk", mWorkDir+"/ramdisk");

if( false == bInit ){
    Log.e("nick", "初始化失败" );
    return retCode;
}

boolean bFixProc = fixer.fix_InitFile_ob();
if( !bFixProc ){
    Log.e("nick","修复失败");
    return retCode;
}

boolean bUninit = fixer.endParse();
if( !bUninit ){
    Log.e("nick","反初始化失败");
    return retCode;
}
```

相关代码(重打包&回写设备)

```
//gzip压缩打回GZIP包
try {
    gzipUtils.compress(mWorkDir+"/ramdisk", false);
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
    return retCode;
}

//回写设备
imgTools img_writer = new imgTools();

long filesize = 0;
File f = new File(mWorkDir+"/ramdisk.gz");
if( !f.exists() ){
    return retCode;
}
//获取重打包后ramdisk.gz文件的大小
filesize = FilesUtils.getFileSize(f);
img.modifyHeaderFile(mWorkDir+"/bootimg.header",
        mWorkDir+"/img.header", filesize );

//回写boot.img头部
SuUtils.execmd("dd if="+mWorkDir+"/img.header" +
        " of=" + bootPartition    +" obs=1 count=608");

//回写ramdisk部分
SuUtils.execmd("dd if="+mWorkDir+"/ramdisk.gz of=" + bootPartition
        +" obs=1 count="+filesize+" seek="+ramoff);
```

代码实现要点回顾

- 1 APK启动的su shell dump出来的文件权限较高，后续返回至APK中处理这些文件时可能会遇到权限问题，可以再调用chmod在APK处理这些文件前修改文件的权限。
- 2 JAVA语言的变量类型和C/C++语言的变量类型不完全相同，混合处理时避免因为语言的差异导致部分变量处理错误。

演示

- 演示
- 工具下载地址:

https://play.google.com/store/apps/details?id=com.netqin.ripper

- Q&A

结束

Thank you!