梆梆专业版加固技术分析
之防篡改

学习群号：211730239

目　　录

文件版本说明

表 1 版本说明

版本	发布时间	修订章节	作者
1.0	2017/12/5	初始版本	oooAooo

参考资料

1. http://blog.csdn.net/jiangwei0910410003/article/details/50402000
2. http://blog.csdn.net/devilcash/article/details/7230733
3. http://download.csdn.net/download/u_1_n_2_i_3/9722166

编写目的

本篇主要主要介绍梆梆安全加固防篡改的原理和方法。选择梆梆，是因为其在防篡改方面是比较突出的。关于 android 签名机制，有兴趣的可以看一下参考资料 1。

声明

本文仅限于技术讨论，不得用于非法途径，后果自负。

名词定义和缩略语说明

表 2 名词定义及缩略语说明

序号	缩写	说明
1	BB	梆梆

1 写在前面

既然我们选择 BB，我们先看一下 BB 在防篡改技术的介绍，下图是 BB 官网上关于防篡改的介绍。（https://www.bangcle.com/products/productindex?product_id=1）

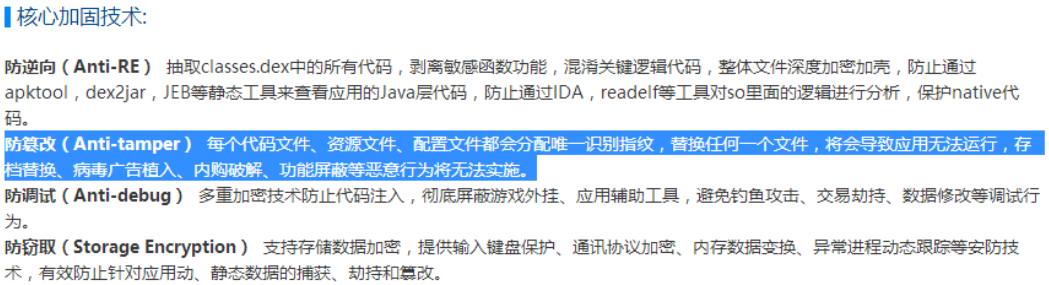

图 1-1 [BB 加固之防篡改功能说明]

从上图可知，APK“任何文件”（这里的任何加引号的意思是 APKl 里面还是有几个文件不在其防篡改的保护之内的，我们后面会说明）被替换或修改，应用都无法运行。实现的方法是文件都会分配一个识别指纹。

根据 BB 这个说明，我们将开始逆向分析其防篡改技术。

实际在逆向分析中我们会发现其采用很多加密算法和数据结构。为了便于大家对了解其防篡改的技术全貌，先介绍分析完后的成果，然后在下一篇介绍这些成果是怎样分析过程。

2 BB 算法与数据结构

BB 在其防篡改实现中用到了很多加密算法和数据结构，下面首先介绍其用到了那些加密算法和数据结构。

2.1 数据结构和算法

2.1.1 使用了如下 6 种加密算法：

- MD5
- RC4
- 斐波那契数列
- BAS64
- SHA-1
- RSA

2.1.2 使用了如下 3 种数据结构

- UTHASH

- 单项链表
- ZIP

参考资料的 2 和 3 是关于 UTHASH 与 sha1withRDA 的介绍，感兴趣的话可以看一下。

2.2 自定义的结构变量

2.2.1 APK\assets\meta-data\ manifest.mf 文件格式

BB 在其防篡改功能说明中说到，会为每个文件分配一个识别指纹，那么这个指纹到底存在哪里呢。实际上 APK\assets\meta-data\ manifest.mf 文件存放的就是对应的指纹信息。下面介绍一下这个文件的格式。

```
j20KiwhQW5kcm9pZE1hbmlmZXN0LnhtbA==sAdB/Mv
SOkYwu0prp/szv4eb6aG3PBhZKsWqkWBKaV9846zox
ff8tXFZ4eMjm8pFXXvD869etVeBlXtHx7dtYC27762
UXM5708mUVbAkiMfRLjIBsyeeaa8bZBhMKHUAv6DUJ
0RsmllQ8f94gYjYLf+lxePOBt5D9c07Oc3gdqGOmH5
icL3QJnWvjA4eb9+/aEORLJw9HWLSSCe287YJXl16s
wYkq8p7Q3NiOzlx7265DgthtPOycz+j8hyc4ec2pQI
dl88rUFVxlvTPLSQIVx5e2ciqYIaMPIg/dFDorie77
NnC4a9eJn4RrlxO/CGXRoifba6a8vLr225bV044ss8
```

图 2-1 [APK\assets\meta-data\ manifest.mf]

从上图中可以看到这个文件的内容看起来像是 BASE64 编码，实际上是不是呢，我们可以验证一下，先将文件开始像 base64 编码复制出来如下：

“j20KiwhQW5kcm9pZE1hbmlmZXN0LnhtbA==”但是这个字符串是 35 个字符，对于 base64 编码一定是 4 字节对齐的，不够则用=补齐。我们可以把前面的 3 个字符去掉，让其是 32 个字节（为啥去掉 3 个字节，实际上在后面逆向分析过程中我们可以明白），经过 base64 解码后，如下：

图 2-2 [APK\assets\meta-data\ manifest.mf 内容解码]

下面就将这个文件的具体格式说明，后面再说明如何知道的。

表 2-1［[APK\assets\meta-data\ manifest.mf 文件格式]

偏移	字节数	说明
0	1	其与 0x68 异或后为 2，表示后面字符串的长度
1	2	表示后面文件名对应的 base64 的长度
2	20	AndroidManifest.xml 字符串的 base64
36	28	文件指纹 3，原始 APK 的所有文件 sha1 的前 4 个字符组合在一起的 sha1 再进行 base64
64	28	实际分析中，并没有用到
92	28	实际分析中，并没有用到
120	28	实际分析中，并没有用到
148	28	文件指纹 4，原始 APK 所有文件的 CRC32 的低 4 个字节组合在一起的 sha1 再进行 base64
176	4	文件指纹 1，原始 APK 压缩包中第一个文件的 sha1 字符串的前 4 个字节
180	4	实际分析中，并没有用到
184	4	实际分析中，并没有用到
188	4	实际分析中，并没有用到
192	4	文件指纹 2，原始 APK 压缩包中第一个文件的 CRC32 的低 4 个字节转换成的字符串。
192+N*20	4	第 N 个文件，依次类推

实际上整个签名校验的过程就是验证上面的 4 个文件指纹，来判断的。在后面的分析中我们会真正的了解。

2.2.2 BB_HashTable

BB_HashTable 主要用于存储 APK\\META-INF\ MANIFEST.MF 文件内容的。
APK\\META-INF\ MANIFEST.MF 文件存储的是 APK 包中所有文件的 SHA1 签名：

```
Manifest-Version: 1.0

Name: AndroidManifest.xml
SHA1-Digest: KEbfcZikzEEld0mfzyxy4COF7AQ=

Name: assets/AZURE.png
SHA1-Digest: lcbV0PUldzpxDYiaXn5ePQDihB0=

Name: assets/App.zip
SHA1-Digest: Vl8E82WIZxngcR59wgfB029ARQU=

Name: assets/BLUE.png
SHA1-Digest: 48pxvOeYGxMU6vlDPSvopC8wKNs=

Name: assets/CYAN.png
SHA1-Digest: DOGgZDZe38eYauGGNAoeYd+tdrk=

Name: assets/GREEN.png
SHA1-Digest: E5Kx0EQK6ZzGdJU7s3yp2cmBG48=
```

图 2-3 [APK\\META-INF\ MANIFEST.MF]

其定义如下：

```
typedef struct BB_HashTable
{
    char* fileName;                 //文件名
    char* sha1;                     //对应文件的sha1签名
    int   noUsed0;
    int   noUsed1;
    int   noUsed2;
    UT_hash_table strhashTalbe;     //uthash结构

}BB_HashTable;
```

其中成员 sha1 用于文件指纹 1 的判断，同时用于组合文件指纹 3。

2.2.3 ORG_file_Sign_list

这个结构主要是保存加固时 APK 的原始签名。如果实现防篡改功能，势必要保存原始 APK 的指纹信息，然后与当前 APK 的指纹信息进行对比，从而来判断是否被篡改。

这个结构对应的数据内容在如下目录：APK\assets\meta-data\ manifest.mf，其定义如下：

```
typedef struct ORG_file_Sign_list
{
    int index;              //链表索引   like 0 1 2 3...
    char* sha1First4Char;   //对应文件sha1字符串20个字符的0-3个字符(指纹1)
    char* sha1Second4Char; //对应文件sha1字符串20个字符的4-7个字符
    char* sha1third4Char;   //对应文件sha1字符串20个字符的8-11个字符
    char* sha1fourth4Char; //对应文件sha1字符串20个字符的12-15个字符
    char* sha1fifth4Char;   //对应文件sha1字符串20个字符的16-19个字符（指纹2）
    ORG_file_Sign_list* next;  // 下一个节点 如果为0则表示是尾节点。
```

```
}ORG_file_Hash_list;

typedef struct ORG_fileSign
{
    //文件指纹3，原始APK的所有文件sha1的前4个字符组合在一起的sha1再进行base64
    char* allFileSha1First4Char;
    char* noUsed0;
    int   noUsed1;
    int   noUsed2;
    //文件指纹4，原始APK压缩包中第一个文件的CRC32的低4个字节转换成的字符串
    char* allFileCrc32Sha1;
    //单个文件对应的文件指纹信息，文件指纹3与文件指纹4
    ORG_file_Sign_list* orgFileSignListHead;
}ORG_fileSign;
```

本文件是识别指纹的核心文件，里面共存储了 4 种类型的文件指纹如下;

- 指纹 1：原始 APK 单个文件 sha1 字符串的前 4 个字节
- 指纹 2：原始 APK 单个文件对应的 CRC32 的低 4 个字节
- 指纹 3：指纹 1 组合起来的 SHA1 的 base64
- 指纹 4：指纹 2 组合起来的 SHA1 的 base64

2.2.4 CUR_file_crc32 结构

本结构用于存储当前 APK 中每个文件的 CRC32 的数值，用于进行文件指纹 2 的比较。

```
typedef struct CUR_file_crc32
{
  char* fileName;                    //文件名
  int   crc32;                       //文件对应的CRC32
  CUR_file_crc32 next;               //下一个节点
}CUR_file_crc32;
```

3 BB 防篡改之文件指纹形成

从上一节中我们可以知道其存在 4 种类型的识别指纹，这些指纹存储在“APK\assets\meta-data\ manifest.mf”中。实际上存在另外 2 个文件见下图：

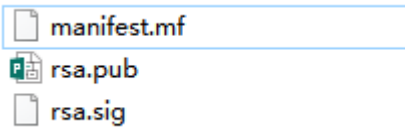

图 3-1 [BB 加固之防篡改识别指纹文件]

Rsa.pub 和 ras.sig 文件到底是干什么的？ras.sig 保存的是 manifest.mf 的 RSA 加密后的

签名。Ras.pub 保存的是 RSA 的公钥信息。

BB 不大可能拿到 APP 开发者签名的私钥，依次这个 ras.pub 应该是 BB 自己的公钥。

这 2 个文件的作用其实就是验证原始的文件指纹是否被篡改。

3.1 文件指纹形成流程

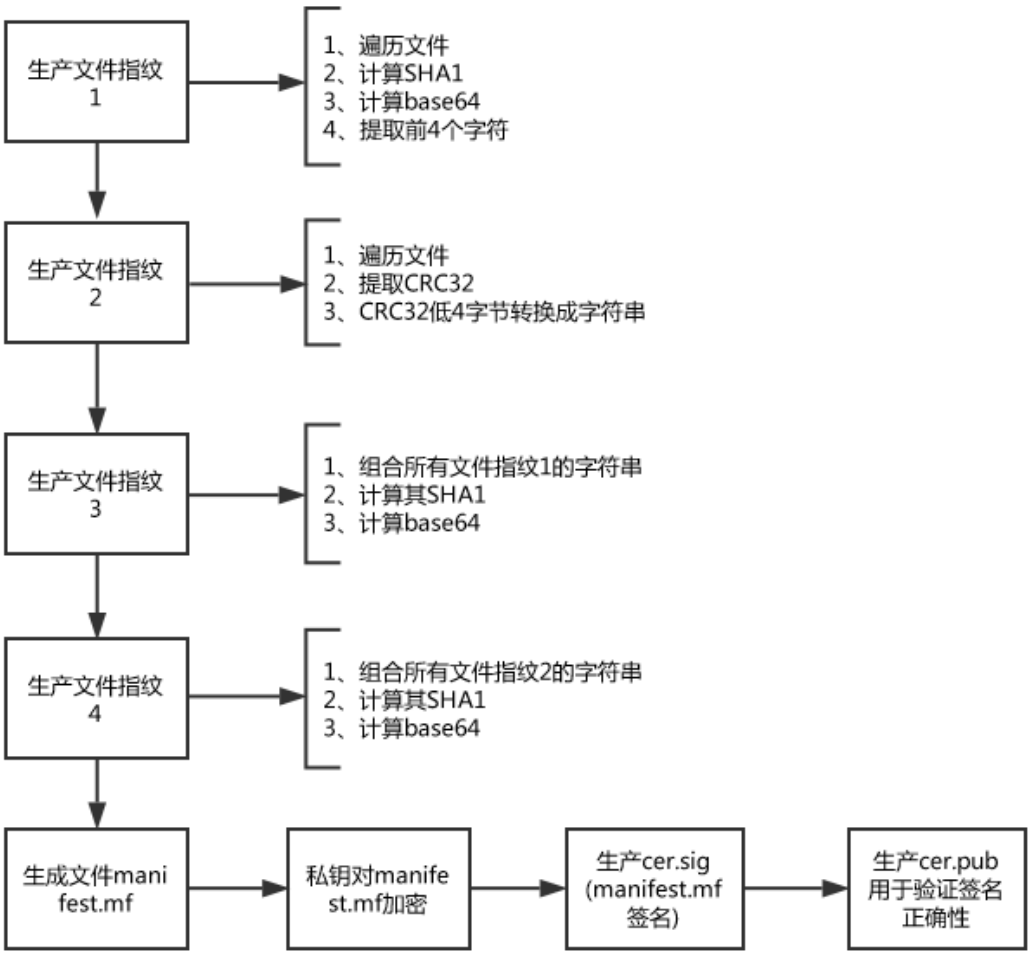

图 3-2 [文件指纹形成流程图]

3 BB 防篡改之验证流程

整个验证流程可以分为如下 8 个步骤：

1、 验证原始 APK 的文件指纹(manifest.mf)、文件指纹的签名(cert.sig)以及公钥文件(cer.pub)是否被篡改，如果被篡改，则终止应用。

2、 提取原始 manifest.mf 的 4 种类型的文件指纹

3、 提取当前文件的 sha1，其中包括当前文件指纹 1

4、 提取当前文件的 CRC32，包括当前文件指纹 2

5、 对当前文件指纹 1 与原始文件指纹 1 比较，如果不一致，则终止应用。

6、 对当前文件指纹 2 与原始文件指纹 2 比较，如果不一致，则终止应用。

7、 将当前 APK 所有文件指纹 1 组合成一个字符串，并计算其整体的 SHA1，然后再 base64，即获得当前 APK 的文件指纹 3，与原始的文件指纹 3 进行比较，如果不一致，则终止应用。

8、 将当前 APK 所有文件指纹 2 组合成一个字符串，并计算其整体的 SHA1，然后再 base64，即获得当前 APK 的文件指纹 4，与原始的文件指纹 4 进行比较，如果不一致，则终止应用。

3.1 验证原始 APK 指纹文件的有效性

原始 APK 文件指纹文件包括如下三个文件：

- manifest.mf：原始 APK 的文件指纹，文件格式见 2.2.1
- cert.sig：manifest.mf 经过 RSA 加密后的签名
- cer.pub：用于验证 cert.sig 的 RSA 公钥。

这里的 RSA 签名，并不是 APK 开发者的签名，而是 BB 自己的签名。

3.1.1 验证 cert.pub 文件正确性

在 BB 的 libdexhelper.so 文件中保存着 cert.pub 的 MD5 值，但是这个 MD5 值是经过加密后的，而且加密的流程还是比较复杂的。

解密流程首先实现了一个从 APK 文件中提取制定文件的 C 语言版本解压相关程序：包括如下函数：

- dexZipOpenArchive //打开 zip 文件获得 ZipArchive 结构
- dexZipFindEntry //总 ZIP 中获取指定文件的入口结构
- dexZipGetEntryInfo //得到指定文件入口结构对应的信息
- ReadToBufFromZipFile //将指定文件的内容读到 BUF 中
- dexZipCloseArchive //关闭 zip 文件

利用上面的函数，将 cert.pub 读取到内存中，然后计算其 MD5 值。下面开始获得获取原始 APK 的 MD5 值，经过如下步骤：

1、计算 0x52600 开始 0x1000 个字节的 MD5 值。

2、构造一个斐波那契数列，1 1 2 3 5 8 13 21 34 55

3、以斐波那契数列为索引从数组 0x52600 取出对应的数值，然后与步骤 1 得到的 MD5 值按字节亦或，获得一个 0x10 大小的 KEY。实际上是下面 RC4 解密的 KEY

4、利用步骤 3 获得的 RC4 KEY ，对 525E0 开始的 0x10 大小的数据，进行 RC4 解密。解密的结果就是原始的 cer.pub 对应的 MD5 值。

然后将 2 个 MD5 值进行比较，如果不一致，则终止应用。

3.1.2 验证 manifest.mf 正确性

经过如下步骤进行验证：

1、 读取 manifest.mf 文件到内存
2、 读取 cert.sig 文件到内存
3、 使用 cert.pub 对 manifest.mf 进行 RSA 签名
4、 利用步骤 3 获得签名与 cert.sig 进行比较，如果不一致，则终止应用。

3.2 提取当前 APK 的文件指纹 1

将 APK\\META-INF\ MANIFEST.MF 读到内存，并进行一行一行的遍历，将文件名和对应的 sha1 数值存储在结构 BB_HashTable 中。BB_HashTable 结构见 2.2.2。

3.3 提取原始 manifest.mf 的 4 种类型的文件指纹

对原始 manifest.mf 文件进行解析，并将解析结果存储到 ORG_file_Sign_list 结构中，ORG_file_Sign_list 结构见 2.2.3

3.4 提取当前文件的文件指纹 2

遍历 APK，过滤掉“assert\ meta-data”、“META-INF\”目录，并且获得每个文件的 CRC32 的数值，并存储在 CUR_file_crc32 结构中，CUR_file_crc32 结构见 2.2.4。

3.5 文件指纹验证

前面已经将原始 APK 的 4 种文件指纹，以及当前 APK 的 4 种文件指纹都获取到，然后就开始进行判断，只要有任何一个文件指纹不一致，则终止应用。

4 终止应用的方法

对于通用的终止应用的方法，一般可能采用 kill、abort、exit 等系统函数，BB 显然知道这样去终止会带来很大的安全问题。因此其采用了另外一种终止方式。

当需要终止应用时，首先破坏堆栈内容，然后将 PC 指针指向存储 APK 路径名的位置，而其对应的代码是无法执行的，从而引发异常来使应用终止运行。

5 总结

1、 提取原始文件的的文件指纹，并分为 4 种类型；
2、 使用 RSA 对原始文件指纹进行加密；
3、 同时在 SO 中保存 RSA 公钥的 MD5 值，而这个 MD5 值是经过多重加密存放的。里面用到了 MD5\斐波那契数列\RC4\base64 等加密算法；

4、 验证 RSA 公钥正确性；
5、 验证原始文件指纹的正确性，使用 RSA 公钥验证；
6、 至此确保原始文件的所有指纹都是正确的，下面开始进行真正的校验过程；
7、 获取原始文件的 4 种文件指纹
8、 获取当前文件的 4 种文件指纹
9、 进行 4 种文件指纹的一一比较，只要有一个不符合，则终止进程。