LEAVING OUR ZIP UNDONE: HOW TO ABUSE ZIP TO DELIVER MALWARE APPS

Gregory R. Panakkal
K7 Computing, India

Email gregory.panakkal@k7computing.com

ABSTRACT

2013 saw multiple high-profile vulnerabilities for *Android*, with the 'Master Key' Cryptographic Signature Verification Bypass vulnerability topping the charts. Several specially crafted malicious APKs exploiting this vulnerability appeared after proof-of-concepts (PoCs) were created by its initial discoverers. It was the difference in the two ZIP archive-handling implementations used by *Android* – one to validate the APK (using Java), and other to extract the contents of the APK (using C) – that led to this vulnerability.

ZIP is the de facto standard packaging format for delivering applications such as Android Package (APK) files, Java Archive (JAR) files, Metro App (APPX) files, and documents such as Office Open Format (DOCX, XLSX etc.) files. Both *Android* and Java malware, delivered via ZIP-based packages, have reached high volumes in the wild, and continue to grow at a rapid rate. Therefore, it is critical for anti-virus engines to scan the contents of these files correctly, matching the behaviour observed in the target environment.

This paper explores the ZIP file format, focusing specifically on APK files as handled by the *Android* OS. It covers the existing design, and technical aspects of publicly disclosed vulnerabilities for *Android*. The paper also explores new malformations that can be applied to APK files to break typical AV engine unarchiving, thus bypassing content scanning, while keeping the APK valid for the *Android* OS. It briefly covers the concept of an amalgamated package ('Chameleon ZIP') that could be treated as an APK/JAR/DOCX file, based on the application that processes it, and the challenges this poses to the AV engine components that attempt to scan content based on recognized package type.

APK PRIMER

Android Package (APK) is a ZIP-based file containing *Android*-specific metadata, with a directory structure and package verification metadata similar to that of JAR (Java Archive). This design makes it easier for the Android Package Manager to locate, extract and validate the contents within the APK. (The following section may be skipped if the reader is familiar with the ZIP file format.)

ZIP file format

The ZIP file format includes a central directory located at the end of the file. The central directory, containing names of files or directories and related information, acts as a reference to easily locate a file's compressed data.

The three primary headers that are used by the ZIP format are as follows:

1. **End of Central Directory (EOCD) header**

   An unarchiving application scans the last 64KB of the file in reverse order to locate the EOCD header's magic value (0x06054b50). Once located, this header provides critical information relating to the location of the start of central directory file headers, the total header size, and the number of entries to expect.

Member	Value (hex)	Size
00042C33 struct EndOfCentralDirectory		00000016
00042C33 SIGNATURE Signature	06054B50	00000004
00042C37 uint16 DiskNumber	0000	00000002
00042C39 uint16 CentralDirectoryStartDisk	0000	00000002
00042C3B uint16 CentralDirectoryStartOffset	0021	00000002
00042C3D uint16 NumEntries	0021	00000002
00042C3F uint32 CentralDirectorySize	00000CE1	00000004
00042C43 uint32 CentralDirectoryOffset	00041F52	00000004
00042C47 uint16 ZipCommentLength	0000	00000002
00042C49 char ZipComment[ZipCommentLength]		00000000

Figure 1: End of Central Directory header.

2. **Central Directory File Header (CDFH)**

   There is one central directory entry per archive file/directory. It consists of critical information such as the filename, the offset to the local header, the size of compressed and uncompressed data, the compression method used, etc. The CDFH starts with the magic value 0x02014B50.

Member	Value (hex)	Size
00042BD6 struct CentralDirectoryFileHeader		0000005D
00042BD6 SIGNATURE Signature	02014B50	00000004
00042BDA VERSION_MADE_BY VersionMadeBy	0014	00000002
00042BDC uint16 VersionNeededToExtract	0014	00000002
00042BDE uint16 GeneralPurposeBitFlag	0808	00000002
00042BE0 COMPRESSION_METHOD Compressio...	0008	00000002
00042BE2 DOSDATE LastModFileTime	A778	00000002
00042BE4 DOSTIME LastModFileDate	42E6	00000002
00042BE6 uint32 Crc32	50EEED0D	00000004
00042BEA uint32 CompressedSize	00008075	00000004
00042BEE uint32 UncompressedSize	0001538C	00000004
00042BF2 uint16 FileNameLength	000B	00000002
00042BF4 uint16 ExtraFieldLength	0024	00000002
00042BF6 uint16 FileCommentLength	0000	00000002
00042BF8 uint16 DiskNumberStart	0000	00000002
00042BFA uint16 InternalFileAttributes	0000	00000002
00042BFC uint32 ExternalFileAttributes	00000000	00000004
00042C00 uint32 RelativeOffsetOfLocalHeader	00039E80	00000004
00042C04 char FileName[FileNameLength]		0000000B
00042C0F blob ExtraField[ExtraFieldLength]		00000024
00042C33 char FileComment[FileCommentLengt...		00000000

Figure 2: Central Directory File Header.

3. **Local File Header (LFH)**

   The Local File Header precedes the file's compressed data, and contains basic information. If specific bits in the GeneralPurposeBitFlag are set, a DataDescriptor structure immediately follows the compressed data to indicate the

CRC32, compressed and uncompressed data sizes. This is typically added by applications that do not know these values at the time of writing the LFH – in which case these fields are set to zero.

Member	Value (hex)	Size
00039E80 struct LocalFileHeader		0000004D
00039E80 SIGNATURE Signature	04034B50	00000004
00039E84 uint16 VersionNeededToExtract	0014	00000002
00039E86 uint16 GeneralPurposeBitFlag	0808	00000002
00039E88 COMPRESSION_METHOD Compressio...	0008	00000002
00039E8A DOSDATE LastModFileTime	A778	00000002
00039E8C DOSTIME LastModFileDate	42E6	00000002
00039E8E uint32 Crc32	00000000	00000004
00039E92 uint32 CompressedSize	00000000	00000004
00039E96 uint32 UncompressedSize	00000000	00000004
00039E9A uint16 FileNameLength	000B	00000002
00039E9C uint16 ExtraFieldLength	0024	00000002
00039E9E char FileName[FileNameLength]		0000000B
00039EA9 blob ExtraField[ExtraFieldLength]		00000024
00039ECD blob FileData[CompressedSize]		00000000

Figure 3: Local File Header.

Member	Value (hex)	Size
0000039E struct DataDescriptor		0000000C
0000039E uint32 Crc32	08074B50	00000004
000003A2 uint32 CompressedSize	739A6403	00000004
000003A6 uint32 UncompressedSize	0000036C	00000004

Figure 4: Data Descriptor Header.

The overall layout of the various ZIP headers and data is shown in Figure 5.

Android Package

The Android Package file follows a predefined directory structure (shown in Figure 6) that enables the Android Package Manager to extract metadata and validate contents before installation.

JAR metadata

Files under the META-INF directory found in the root of the archive enable the Android Package Manager to validate files found outside of this directory.

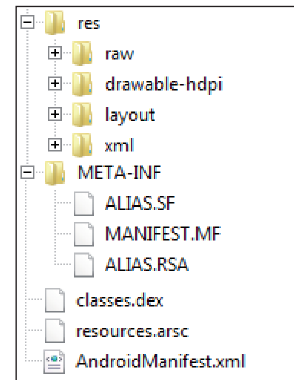


Figure 6: Typical APK directory structure.

- MANIFEST.MF: contains Base64-encoded SHA1 hashes of files with their relative location in the archive.
- *<CERT>*.SF: contains a Base64-encoded SHA1 hash of MANIFEST.MF, and separate Base64-encoded SHA1 hashes of the file entries in MANIFEST.MF.
- *<CERT>*.RSA: certificate file in X.509 format containing the developer's public key and the signed blob of *<CERT>*.SF.

Android metadata

Prior to app installation, the Android Package Manager looks for specific files in the package that provide critical information and are required for the functioning of the app.

- AndroidManifest.xml: A binary XML file specifying essential information about the app to the *Android* system, i.e. information the system must have before it can run any of the app's code. This includes information such as the name of the app, the permissions required, libraries required, etc.
- Classes.dex: A heavily optimized binary in DEX file format, encompassing the compiled Java classes, strings and data into one single executable file.

APK verification and installation

The *Android* installation process initiated by the user involves three main components of the *Android* system.

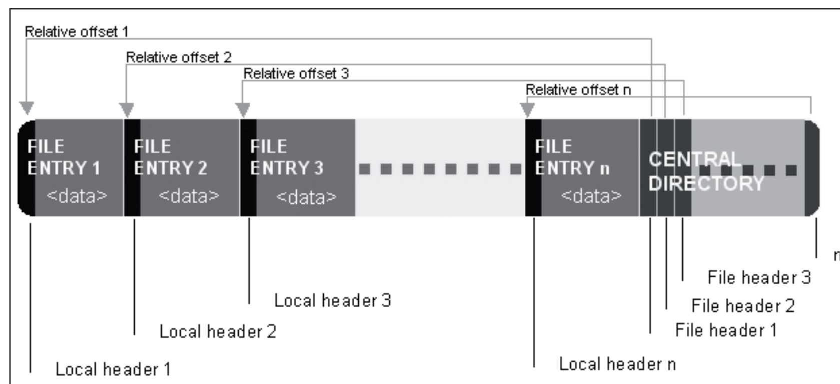


Figure 5: ZIP file layout [1].

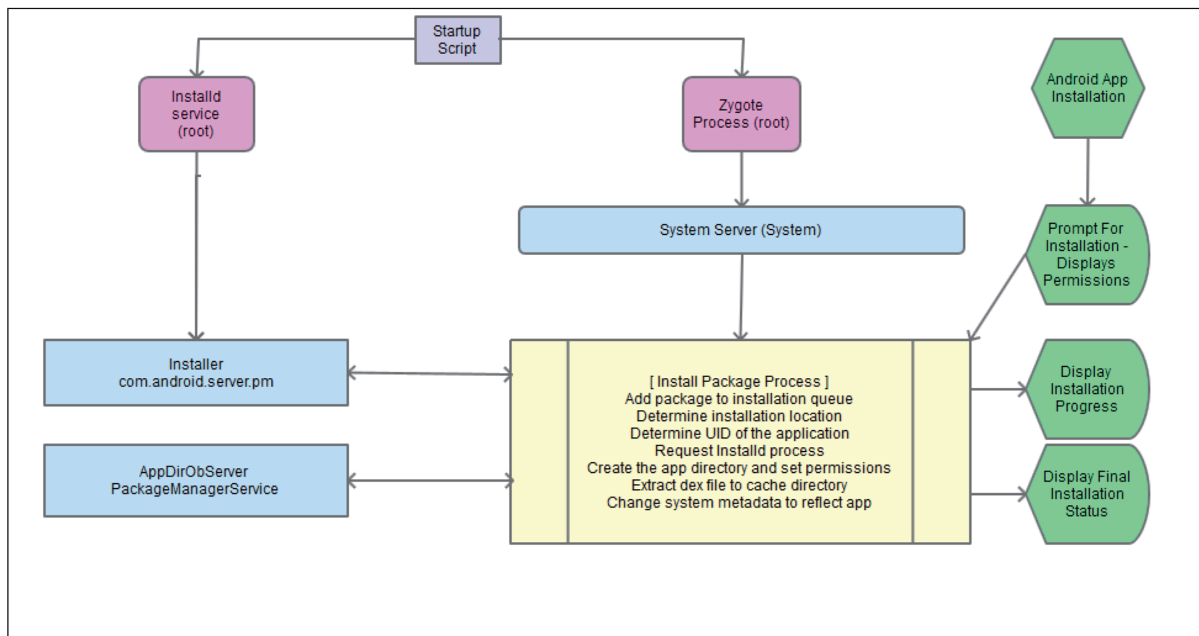


Figure 7: APK installation flow.

Installd

This is a daemon process that runs as root and listens on domain socket /dev/socket/installd for commands to install the APK with appropriate permissions. This component is a native program written in C/C++, and configured to start automatically on OS startup.

Package Manager Service

This runs on startup as part of the system_service process. This service, written in Java, listens for app installation intent. Once it receives a request, it cryptographically verifies the APK prior to invoking the Installd process to perform the on-disk installation.

Package Installer

Package Installer allows the user to install the *Android* application interactively. It communicates with the Package Manager Service to install the application using the Installd process (see Figure 7).

ANDROID CAUGHT WITH ITS FLY DOWN

The *Android* system uses two different implementations when processing an APK file. During verification, it uses the Java implementation (Package Manager Service), and for final extraction it uses a C++ based implementation (Installd/libdex). This has led to breaches in the APK's cryptographic verification process, allowing installation of crafted trojanized APKs without breaking trust.

The first vulnerability abusing the use of multiple ZIP-archive-handling implementations was discovered by *Bluebox Security* and publicized as the 'Master Key' vulnerability [2]. A number of similar vulnerabilities have been discovered since then. The bugs, and their exploitation, have been documented in various technical blogs (mentioned in the References section in detail). The key details of the three vulnerabilities that led to breaches of trust are briefly described below.

Master Key vulnerability – Bluebox Security (BugId: #8219321)

This vulnerability arises from the way in which the two ZIP-handling modules (Java and C++) handle the occurrence of multiple files with the same name. It affects *Android* OS versions prior to 4.3 (*JellyBean*).

The Java implementation enumerates the ZIP file's central directory file header and adds each entry to a LinkedHashMap. The filename is used as the key for the LinkedHashMap. In order to validate the ZIP entries, the Package Manager enumerates through the LinkedHashMap entries. If more than one entry with the same filename is added to the LinkedHashMap, it replaces the previous value. This means that only the last entry for a particular filename in the ZIP is validated.

The C++ implementation, used by the libdex (Installd) when encountering more than one entry with the same name, simply appends it to an in-memory data structure. An unchained hashtable with linear probing is used as the lookup algorithm. So, in this case, when classes.dex is required to be extracted by the VM, the first matching entry is returned. This entry has never been validated by the JarVerifier.

Negative ExtraData signature bypass – Android Security Squad (BugId: #9695860)

The ZIP implementations processing the Local File Headers skip the FileName and ExtraField lengths specified in the LFH

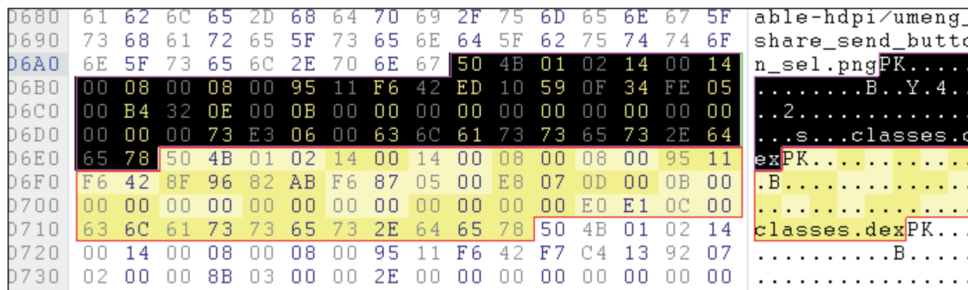
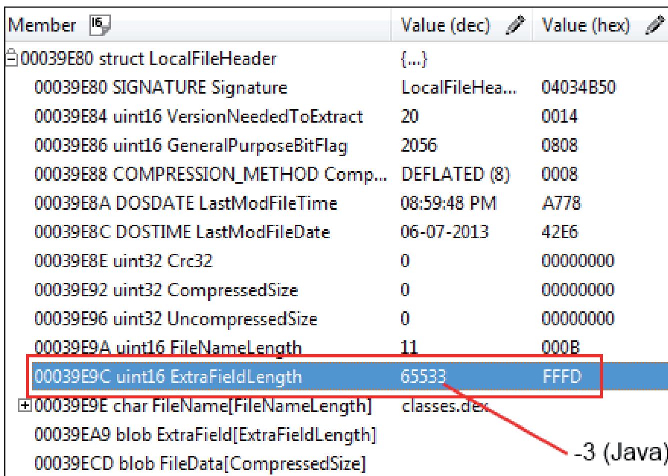


Figure 8: Malicious 'Master Key' APK with two classses.dex central directory entries.

to locate the start of the compressed data. The vulnerability in *Android* OS versions prior to 4.4 (*KitKat*) is that the Java implementation treated this field as a signed integer, while the C++ implementation treated it correctly as an unsigned integer. If a very large unsigned integer is specified as the ExtraData size, it gets interpreted by the Java layer as a negative value. For example, an ExtraData size of 65,533 (0xFFFD) is interpreted as -3. So, while attempting to locate the start of the data, the read pointer ends up moving backwards. On the contrary, the C++ layer would jump forwards about 64KB to locate the data. This discrepancy in the two implementations can be exploited if one locates an APK containing a classes.dex file that is less than 64KB. Such a file (with padding as necessary) can be placed in the area following the LFH and the real file data. The Java layer ends up verifying the benign data, while the attacker-controlled classes.dex is extracted during installation.

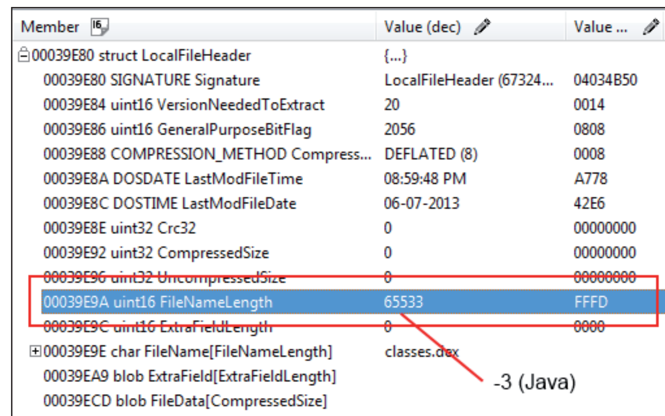


Member	Value (dec)	Value (hex)
00039E80 struct LocalFileHeader	{...}	
00039E80 SIGNATURE Signature	LocalFileHea...	04034B50
00039E84 uint16 VersionNeededToExtract	20	0014
00039E86 uint16 GeneralPurposeBitFlag	2056	0808
00039E88 COMPRESSION_METHOD Comp...	DEFLATED (8)	0008
00039E8A DOSDATE LastModFileTime	08:59:48 PM	A778
00039E8C DOSTIME LastModFileDate	06-07-2013	42E6
00039E8E uint32 Crc32	0	00000000
00039E92 uint32 CompressedSize	0	00000000
00039E96 uint32 UncompressedSize	0	00000000
00039E9A uint16 FileNameLength	11	000B
00039E9C uint16 ExtraFieldLength	65533	FFFD
00039E9E char FileName[FileNameLength]	classes.dex	
00039EA9 blob ExtraField[ExtraFieldLength]		
00039ECD blob FileData[CompressedSize]		

Figure 9: APK with crafted ExtraData field.

Negative FileNameLen signature bypass – Jay Freeman (BugId: # 9950697)

This vulnerability is very similar to the one described above, with signed/unsigned treatment of the FileNameLen field in the Local File Header (see Figure 10).



Member	Value (dec)	Value ...
00039E80 struct LocalFileHeader	{...}	
00039E80 SIGNATURE Signature	LocalFileHeader (67324...	04034B50
00039E84 uint16 VersionNeededToExtract	20	0014
00039E86 uint16 GeneralPurposeBitFlag	2056	0808
00039E88 COMPRESSION_METHOD Compress...	DEFLATED (8)	0008
00039E8A DOSDATE LastModFileTime	08:59:48 PM	A778
00039E8C DOSTIME LastModFileDate	06-07-2013	42E6
00039E8E uint32 Crc32	0	00000000
00039E92 uint32 CompressedSize	0	00000000
00039E96 uint32 UncompressedSize	0	00000000
00039E9A uint16 FileNameLength	65533	FFFD
00039E9C uint16 ExtraFieldLength	0	0000
00039E9E char FileName[FileNameLength]	classes.dex	
00039EA9 blob ExtraField[ExtraFieldLength]		
00039ECD blob FileData[CompressedSize]		

Figure 10: APK with crafted FileNameLen field.

ANDROID APK COMPRESSION METHODS

The ZIP file format specifies at per-file level the compression algorithm (method) that the archiver has used to compress the file. This is a 16-bit field, i.e. 65,536 methods can be defined. *Android*'s application package file (APK), which uses the ZIP format, supports only two compression methods. One is without any compression, i.e. the STORED method (0x0000), and the other is the DEFLATE (0x0008) compression algorithm.

The *Android* OS makes certain assumptions when handling this field, and processes the 'compressed' content based on these assumptions. The pseudocode is given in Table 1.

Android version	C++ ZIP handling	Java ZIP handling
Android 4.4 and above	if Method=Stored RawDataCopy(...) else InflateAndCopy(...)	if Method=Stored RawDataCopy(...) else InflateAndCopy(...)
Android 4.3 and below	if Method=Stored RawDataCopy(...) else InflateAndCopy(...)	if Method=Deflate InflateAndCopy(...) else RawDataCopy(...)

Table 1: Pseudocode on processing 'compressed' content. (Refer to the Appendix for relevant Android ZIP-handling code snippets.)

In most cases, *Android* ZIP handling assumes the compression method to be DEFLATE if the method specified does not match with STORED. In earlier versions of *Android* (4.3 and below),

Java ZIP-handling code checks against the method being DEFLATE, and assumes that the STORED method has been used if it does not match.

ANTI-VIRUS SCANNING AND APK MALFORMATION

Anti-virus software typically handles archive file formats with more stringent checks than any unarchiver utility, or in this case the *Android* OS ZIP-handling layer. This behaviour can be abused by malicious files in order to circumvent the unarchiving process performed by the anti-virus scanning module.

A crafted APK can be constructed primarily by changing the compression method to a value other than STORED or DEFLATE. In this case, the *Android* OS will continue to treat it as either STORED or DEFLATE, but the AV scanner's unarchiving module will be broken when attempting to process the compressed data as per the UNKNOWN method, thereby failing to detect the contents within the APK.

For the purpose of observing the effects of this malformation, repackaged APKs of DroidSheep [3] (a potentially unwanted *Android* app) were used to prepare PoCs. A significant number of anti-virus vendors detect versions of this app (typically on classes.dex). Table 2 shows the behaviour observed.

File name	Android OS <= 4.3	Android OS >= 4.4
Droidsheep_v15_DetectCheck.apk	Install – SUCCESS AV detection – SUCCESS	Install – SUCCESS AV detection – SUCCESS
Droidsheep_v15_Crafted_43.apk	Install – SUCCESS AV detection – FAILED	Install – FAILED AV detection – FAILED
Droidsheep_v15_Crafted_44.apk	Install – FAILED AV detection – FAILED	Install – SUCCESS AV detection – FAILED

Table 2: Behaviour observed with the original and repackaged versions of DroidSheep.

Figures 11 and 12 show Droidsheep_v15_DetectCheck.apk (the original APK without any modifications). The ZIP's central directory header for classes.dex is highlighted, along with the compression method (Figure 12).

Figures 13 and 14 show Droidsheep_v15_Crafted_44.apk (APK crafted to work with *Android* 4.4 and above). Here, changing the COMPRESSION_METHOD to SHRUNK (0x0001) allowed the data still to be treated as DEFLATED, and thus for installation to proceed on the *Android* OS. The method was modified to SHRUNK in both the ZIP's local file header and the central directory header.

Finally, we look at Droidsheep_v15_Crafted_43.apk (APK crafted to work with *Android* 4.3 and below). It is trickier to construct an installable APK with an unhandled compression method considering the mutual reversal of checks in the C++ and Java ZIP-handling layers.

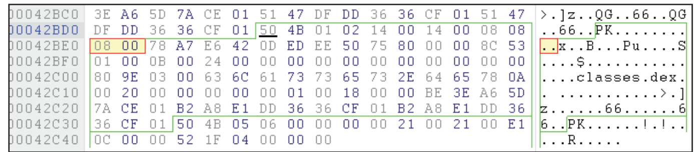
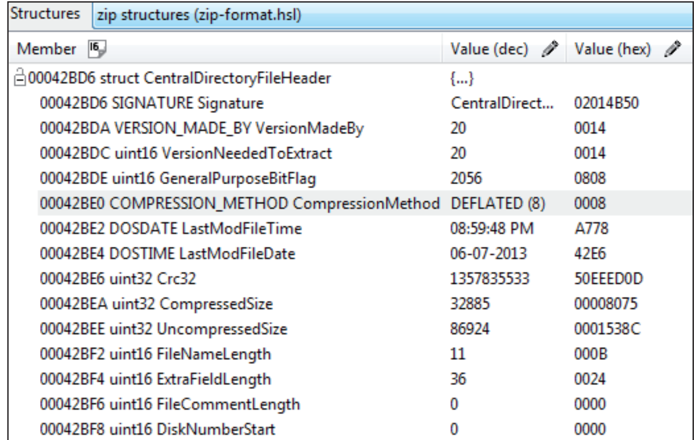


Figure 11: Droidsheep_v15_DetectCheck.apk bytes.



Structures | zip structures (zip-format.hsl)

Member	Value (dec)	Value (hex)
00042BD6 struct CentralDirectoryFileHeader	{...}	
00042BD6 SIGNATURE Signature	CentralDirect...	02014B50
00042BDA VERSION_MADE_BY VersionMadeBy	20	0014
00042BDC uint16 VersionNeededToExtract	20	0014
00042BDE uint16 GeneralPurposeBitFlag	2056	0808
00042BE0 COMPRESSION_METHOD CompressionMethod	DEFLATED (8)	0008
00042BE2 DOSDATE LastModFileTime	08:59:48 PM	A778
00042BE4 DOSTIME LastModFileDate	06-07-2013	42E6
00042BE6 uint32 Crc32	1357835533	50EEED0D
00042BEA uint32 CompressedSize	32885	00008075
00042BEE uint32 UncompressedSize	86924	0001538C
00042BF2 uint16 FileNameLength	11	000B
00042BF4 uint16 ExtraFieldLength	36	0024
00042BF6 uint16 FileCommentLength	0	0000
00042BF8 uint16 DiskNumberStart	0	0000

Figure 12: Droidsheep_v15_DetectCheck.apk fields.

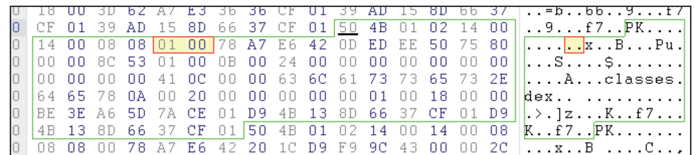
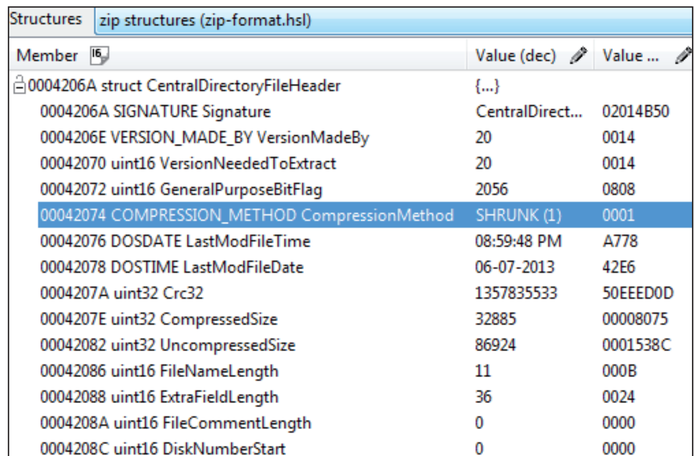


Figure 13: Droidsheep_v15_Crafted_44.apk bytes.



Structures | zip structures (zip-format.hsl)

Member	Value (dec)	Value ...
0004206A struct CentralDirectoryFileHeader	{...}	
0004206A SIGNATURE Signature	CentralDirect...	02014B50
0004206E VERSION_MADE_BY VersionMadeBy	20	0014
00042070 uint16 VersionNeededToExtract	20	0014
00042072 uint16 GeneralPurposeBitFlag	2056	0808
00042074 COMPRESSION_METHOD CompressionMethod	SHRUNK (1)	0001
00042076 DOSDATE LastModFileTime	08:59:48 PM	A778
00042078 DOSTIME LastModFileDate	06-07-2013	42E6
0004207A uint32 Crc32	1357835533	50EEED0D
0004207E uint32 CompressedSize	32885	00008075
00042082 uint32 UncompressedSize	86924	0001538C
00042086 uint16 FileNameLength	11	000B
00042088 uint16 ExtraFieldLength	36	0024
0004208A uint16 FileCommentLength	0	0000
0004208C uint16 DiskNumberStart	0	0000

Figure 14: Droidsheep_v15_Crafted_44.apk fields.

Crafting an installable APK involved the following steps:

Step 1: The directory structure required to construct the APK was prepared (see Figure 15).

Step 2: The classes.dex file to be included in the APK was compressed using an in-house-developed zlib deflate tool. The tool skips writing the zlib headers in the output compressed file,

Name	Size
APKView	487.2 KB
res	343.0 KB
raw	259.9 KB
droidsheep	114.2 KB
droidsheep_bak	114.2 KB
arpspoof	31.5 KB
drawable-hdpi	65.1 KB
layout	10.9 KB
xml	7.1 KB
<Files>	144.2 KB
classes.dex	84.9 KB
resources.arsc	55.3 KB
AndroidManifest.xml	4.0 KB
META-INF	0

Figure 15: Directory layout before compressing classes.dex.

Name	Size
APKView	434.4 KB
res	343.0 KB
raw	259.9 KB
droidsheep	114.2 KB
droidsheep_bak	114.2 KB
arpspoof	31.5 KB
drawable-hdpi	65.1 KB
layout	10.9 KB
xml	7.1 KB
<Files>	91.4 KB
resources.arsc	55.3 KB
classes.dex	32.1 KB
AndroidManifest.xml	4.0 KB
META-INF	0

Figure 16: Directory layout after compressing classes.dex.

making it compatible with unzip's inflate() function. The uncompressed version of classes.dex was replaced in the directory with the compressed version (Figure 16).

Step 3: The contents of the directory structure were then ZIP'ed in STORED mode. This resulted in all the files being packed without any compression (Figure 17).

Step 4: The APK was signed using the JarSigner tool. This resulted in the MANIFEST.MF and related files required for validation containing the SHA1 of the compressed classes.dex (rather than its uncompressed version, as is typically the case).

Step 5: The APK's Central Directory Header and Local File Header then needed to be suitably modified, with the compression method changed to a value other than STORED or DEFLATE. In this case, we used the SHRUNK (0x0001) method. The uncompressed size was also changed to reflect the original uncompressed size of the classes.dex file (see Figure 18).

Installation of this crafted APK works on the target *Android* OS by satisfying the following criteria:

Java verification layer: This layer assumes that if the compression method specified is not DEFLATE, it is STORED. So when it encounters classes.dex with SHRUNK compression, it assumes the STORED method has been used, and matches the SHA1 hash (of the compressed data) against the one specified in the MANIFEST.MF file.

Member	Value (dec)	Value (hex)
0006DB67 struct CentralDirectoryFileHeader	{...}	
0006DB67 SIGNATURE Signature	CentralDirector...	02014B50
0006DB6B VERSION_MADE_BY VersionMadeBy	31	001F
0006DB6D uint16 VersionNeededToExtract	10	000A
0006DB6F uint16 GeneralPurposeBitFlag	0	0000
0006DB71 COMPRESSION_METHOD Compressio...	STORED (0)	0000
0006DB73 DOSDATE LastModFileTime	10:24:38 PM	B313
0006DB75 DOSTIME LastModFileDate	02-03-2014	4462
0006DB77 uint32 Crc32	4232675189	FC497F75
0006DB7B uint32 CompressedSize	32885	00008075
0006DB7F uint32 UncompressedSize	32885	00008075
0006DB83 uint16 FileNameLength	11	000B
0006DB85 uint16 ExtraFieldLength	36	0024
0006DB87 uint16 FileCommentLength	0	0000
0006DB89 uint16 DiskNumberStart	0	0000
0006DB8B uint16 InternalFileAttributes	0	0000
0006DB8D uint32 ExternalFileAttributes	32	00000020
0006DB91 uint32 RelativeOffsetOfLocalHeader	413446	00064F06
0006DB95 char FileName[FileNameLength]	classes.dex	

Figure 17: Classes.dex 'compressed' using the STORED method.

Member	Value (dec)	Value (hex)
0006EC58 struct CentralDirectoryFileHeader	{...}	
0006EC58 SIGNATURE Signature	CentralDirect...	02014B50
0006EC5C VERSION_MADE_BY VersionMadeBy	31	001F
0006EC5E uint16 VersionNeededToExtract	10	000A
0006EC60 uint16 GeneralPurposeBitFlag	2048	0800
0006EC62 COMPRESSION_METHOD Compression...	SHRUNK (1)	0001
0006EC64 DOSDATE LastModFileTime	10:24:38 PM	B313
0006EC66 DOSTIME LastModFileDate	02-03-2014	4462
0006EC68 uint32 Crc32	4232675189	FC497F75
0006EC6C uint32 CompressedSize	32885	00008075
0006EC70 uint32 UncompressedSize	86924	0001538C
0006EC74 uint16 FileNameLength	11	000B
0006EC76 uint16 ExtraFieldLength	36	0024
0006EC78 uint16 FileCommentLength	0	0000
0006EC7A uint16 DiskNumberStart	0	0000
0006EC7C uint16 InternalFileAttributes	0	0000
0006EC7E uint32 ExternalFileAttributes	0	00000000
0006EC82 uint32 RelativeOffsetOfLocalHeader	417554	00065F12
0006EC86 char FileName[FileNameLength]	classes.dex	

Figure 18: Crafted fields for classes.dex data.

C++ ZIP layer: This layer assumes that if the compression method specified is not STORED, it is DEFLATE. So, when it encounters classes.dex with SHRUNK compression, it assumes the DEFLATE method has been used, extracting classes.dex to disk by uncompressing the pre-compressed data.

MITIGATION

Suggested fix for Android OS developers

Android OS should place a stricter check on the compression method fields in order to block the installation of crafted APKs. This has been logged as Issue #69184.

Suggested fix for anti-virus vendors

Having identified the file as APK, anti-virus engines should

choose either to heuristically flag the file if any unsupported compression method is specified in the Local File or Central Directory Header, or to extract the files based on assumptions similar to the ones implemented by the *Android* OS.

CHAMELEON ZIP

The ZIP file format forms the basis for various application packages, including Android Package (APK), Java Archive (JAR), Metro App (APPX) and Microsoft documents (DOCX) etc., which has created new challenges for the AV industry in recognizing the type of packages based on content. A package containing content from various package formats could be treated as APK/JAR/DOCX based on the application that processes it. Identifying the correct package type is critical for any automated analysis system that a security vendor might employ. However, an anti-virus scanner that defaults to extracting the ZIP contents as it sees it will not be affected by this concept package.

ZIP packages are typically checked for files with specific names at specific locations to help identify the package type.

Format	Filenames
JAR	META-INF/MANIFEST.MF
	META-INF/*.SF
	META-INF/*RSA
	*.class
APK	META-INF/MANIFEST.MF
	META-INF/*.SF
	META-INF/*RSA
	AndroidManifest.xml
	classes.dex
DOCX	[Content_Types].xml
	Word
	docProps
	_rels
APPX	AppxManifest.xml
	AppxBlockMap.xml

Table 3: Packages are checked for specific filenames at specific locations.

Relevant file extensions, if available, are usually considered when making a decision on the package type. However, this may not always be the case.

An amalgamated package can easily be created that is valid (with appropriate extension) for the application that processes it (see Figure 19).

The same package can be installed as an *Android* app (with .apk extension), run as a regular Java app/applet (with .jar extension), opened by a document processor (with .docx extension), etc.

Figure 20 shows the directory tree structure from which we created the Chameleon ZIP. It contains a mix of files that are all part of the various file formats.

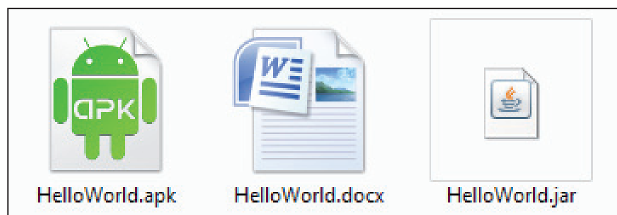


Figure 19: A few of the popular ZIP-based formats.

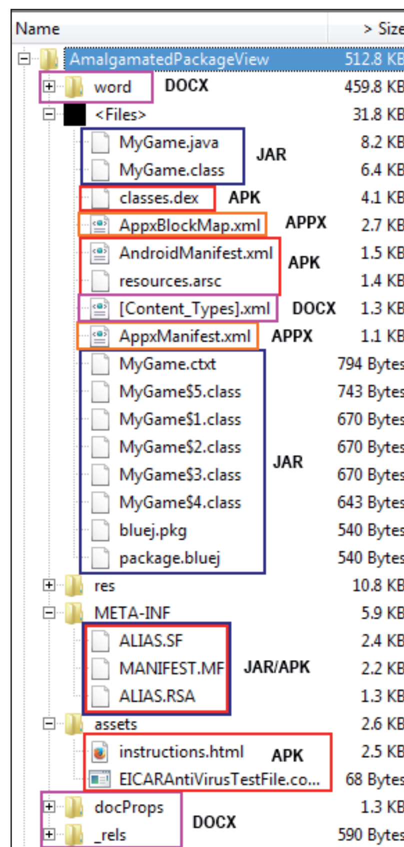


Figure 20: Sample Chameleon ZIP layout.

The APK and JAR files follow the same signing process, which makes them valid for the respective applications. The *Android* OS and JAR treat the irrelevant files just as other data files.

We observed that *Microsoft Word* has a requirement that it is able to identify and relate to all files within the amalgamated package renamed to DOCX. Therefore, attempting to open it in *Word* fails, with a message declaring the file to be corrupt. However, *OpenOffice* was able to successfully open and display the content without any errors or warnings.

Windows 8 Metro apps also require each known file's integrity to be verified, so it may not be possible to create a valid APPX and APK/JAR due to the fact that the two latter formats require updating of independent files with hashes needed for validation. However, having the APPX-related files within the APK/JAR might be enough to throw off a package type analysing component.

Format/extension	Application	Status
APK	Android OS	Success
JAR	Java Runtime	Success
DOCX	OpenOffice	Success
DOCX	Microsoft Word	Failed
APPX	Windows 8	Failed

Table 4: Chameleon ZIP status.

ZIPPING IT UP

Owing to its flexibility, versatility and popular use, the ZIP format has been the target of many types of manipulation in order to bypass anti-virus scanning and OS validation. Based on the history of the format's misuse, and the multitude of major implementations that process the format, we expect this trend to continue. It is best that the anti-virus vendors remain on their toes.

Our internal analysis showed that a crafted APK with non-conformant compression method was affecting various anti-virus vendors. Both their mobile and *Windows* products were affected. The technical details have been shared with the respective vendors.

The effect of Chameleon ZIP on automated systems was not evaluated due to practical restrictions. We suggest that the anti-virus vendors evaluate their own systems based on the information provided in this paper.

REFERENCES

[1] Sourced from http://en.wikipedia.org/wiki/ZIP_%28file_format%29.

[2] https://bluebox.com/technical/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/.

[3] http://www.droidsheepapk.com/.

[4] http://www.saurik.com/.

[5] Parmar, K. http://www.kpbird.com/2012/10/in-depth-android-package-manager-and.html.

[6] Mody, S. I am not the D ’r.0,1d You are Looking For: An Analysis of Android Malware Obfuscation. Proceedings of the Virus Bulletin International Conference 2013.

[7] Dhanalakshmi, V. How vulnerable is Android to attack? AVAR 2013.

[8] https://android.googlesource.com/.

APPENDIX: ANDROID ZIP-HANDLING CODE SNIPPETS

For Android OS >= 4.4 (after Master Key fixes):

C++ Source @ https://android.googlesource.com/platform/dalvik.git/+/android-4.4.2_r2/libdex/ZIPArchive.cpp

```
if (method == kCompressStored) {
   if (sysCopyFileToFile(fd, pArchive->mFd, uncompLen)
!= 0)
      goto bail;
} else {
   if (inflateToFile(fd, pArchive->mFd, uncompLen,
compLen) != 0)
      goto bail;
}
```

Java Source @ https://android.googlesource.com/platform/libcore.git/+/android-4.4.2_r2/luni/src/main/java/java/util/zip/ZIPFile.java

```
if (entry.compressionMethod == ZIPEntry.STORED) {
   rafStream.endOffset = rafStream.offset + entry.
size;
   return rafStream;
} else {
   rafStream.endOffset = rafStream.offset + entry.
compressedSize;
   int bufSize = Math.max(1024, (int) Math.min(entry.
getSize(), 65535L));
   return new ZIPInflaterInputStream(rafStream, new
Inflater(true), bufSize, entry);
}
```

For Android OS <= 4.3:

C++ Source @ https://android.googlesource.com/platform/dalvik.git/+/android-4.2.2_r1/libdex/ZIPArchive.cpp

```
if (method == kCompressStored) {
   if (sysCopyFileToFile(fd, pArchive->mFd, uncompLen)
!= 0)
      goto bail;
} else {
   if (inflateToFile(fd, pArchive->mFd, uncompLen,
compLen) != 0)
      goto bail;
}
```

Java Source @ https://android.googlesource.com/platform/libcore.git/+/android-4.2.2_r1/luni/src/main/java/java/util/zip/ZIPFile.java

```
if (entry.compressionMethod == ZIPEntry.DEFLATED) {
   int bufSize = Math.max(1024, (int)Math.min(entry.
getSize(), 65535L));
   return new ZIPInflaterInputStream(rafstrm, new
Inflater(true), bufSize, entry);
} else {
   return rafstrm;
```