

ZIP It Up, Sneak It In: Revealing evasion techniques in Android malware

Leonidas Vasileiadis

leonidas.vasileiadis@secura.com

Kaloyan Velikov

kaloyan.velikov@secura.com

April 2024

Contents

1	Introduction	2
2	Background	4
2.1	The ZIP format	4
2.1.1	General ZIP handling	5
2.2	APK installation	5
2.2.1	Extraction of the AndroidManifest.xml	6
2.2.2	Parsing of the AndroidManifest.xml	11
3	Problem Definition	21
3.1	Exploiting the ZIP structure	21
3.1.1	Tampered compression methods	21
3.1.2	Zipentry with empty filename	24
3.2	Exploiting the AXML structure	25
3.2.1	Spoofing the Type Identifier	25
3.2.2	Tampered stringCount value	26
3.2.3	Strings surpassing maximum length	27
3.2.4	Invalid data between elements	28
3.2.5	Unexpected attribute size	29
3.2.6	Unexpected attribute names or values	30
3.2.7	Zero size header for namespace end nodes	31
4	Approach	32
4.1	Collection of APKs	32
4.2	Current Tooling Landscape	33
4.2.1	Jadx	33
4.2.2	Apktool	34
4.2.3	ApkAnalyzer	34
4.2.4	Androguard	34

4.3	Empirical Testing Methodology	34
4.4	Development of the New Tool	35
4.4.1	Empirical Testing Methodology	36
4.4.2	Integration with Existing Tooling	37
5	Empirical Validation	38
5.1	Current Tooling	38
5.1.1	jadx	38
5.1.2	apktool	40
5.1.3	apk-analyzer	41
5.1.4	Androguard	43
5.2	apkInspector	44
5.2.1	Three-Stage Evasion Technique Testing	45
5.2.2	Play Store Application Testing for Accuracy	46
6	Conclusions	48
6.1	Future Work	48
	Bibliography	50

Abstract

As of 2024, the Android community faces a growing challenge of advanced evasion techniques being employed by an increasing volume of Android malware to bypass static analysis. These techniques not only allow these applications to infiltrate the Google Play Store, amassing thousands of unsuspecting users, but also evade detection by online analysis providers. Such applications, potentially part of modern malware campaigns, expose a critical vulnerability in our digital ecosystem—our inability to detect and mitigate threats that cleverly disguise their malicious intentions. This flaw compromises the security and privacy of individual users and undermines trust in our app marketplaces.

In response to these challenges, this paper provides detailed description about the evasion tactics employed by modern Android malware and introduces a novel tool specifically designed to identify and remedy these evasion techniques. With the ability to detect and analyze advanced evasive malware, our tool plays a crucial role in restoring the transparency and resilience of the Android ecosystem.

1. Introduction

Malware like Spynote (February 2024 [4]) or Soumnibot (April 2024 [5]) actively use evasive methods to bypass certain AV detections and remain undetected for longer and continue spreading. While actively exploited nowadays, some of these approaches are not entirely new on the malware detection evasion scene. Sources as early as 2013 [3] mention that manipulations of the ZIP archive structure were employed to interrupt the AV engines' decompression functions, resulting in insufficient scanning results. Despite creating a practically invalid ZIP file according to the ZIP specification [1], these Android installation files (APK) remain valid and could be installed to devices or distributed via alternative app stores. This technique is actively used by malware developers [7] to provide valuable time before their creation gets detected and taken down.

Over time, it has become evident that both online analysis providers and community driven tools continue to lack the capability to effectively parse APK files that employ techniques to circumvent traditional static analysis detection methods. This deficiency highlights a persistent gap in security measures, where it is not possible to properly assess and mitigate the risks posed by such files. Furthermore, we are observing an increase in the number of Android malware deploying such evasion techniques, possibly driven by the emergence of a semi-automated malware-as-a-service business model in the underground[2]. This makes sophisticated evasion capabilities more accessible and widespread, further challenging existing security frameworks.

Our approach involved analyzing active malware applications from which we identified nine distinct evasive tactics which were examined and explained in detail through the Android source code. Recognizing the necessity for enhancing static analysis efforts among malware researchers, we have developed a versatile tool serving both as a command-line interface (CLI) and library combined. This tool serves to aid in static analysis and can seamlessly integrate into existing solutions. Our end goal is to enhance the detection of tampered APK files and provide robust solutions for in-depth exploration of their contents.

The rest of the paper is organized as follows. Section 2 provides details along with the relevant source to prepare the reader with the necessary knowledge in order to understand the next chapter. Section 3 states the problem that we are addressing. Then, Section 4 covers our methodology of identifying evasion tactics. Our empirical validation is presented in Section 5. Section 6 closes the paper.

2. Background

2.1 The ZIP format

The APK (Android Package) file format, at its core, is essentially a ZIP archive. This fundamental aspect of its design means that any tool capable of decompressing ZIP files should theoretically be able to open and extract the contents of an APK. Whether it is popular software like WinZip, 7-Zip, or built-in operating system utilities, these tools can readily access the contents of an APK, treating it like any other ZIP archive. This inherent compatibility simplifies the process of working with APK files, allowing developers, researchers, and users to access, extract, and analyze the contents using a wide range of familiar tools.

Within a general ZIP file, the entries can be compressed using various algorithms. However, for Android APKs, only two compression algorithms are typically employed: deflated and stored. Deflated compression, a common algorithm used in ZIP files, reduces the size of stored data by identifying and eliminating redundancy. Conversely, the "stored" method simply stores the data without compression. This selective use of compression algorithms within APKs ensures optimal performance and compatibility across Android devices while maintaining efficient storage and distribution of application resources.

In ZIP files, the central directory header serves as a crucial component containing essential information about all entries in the archive. This directory header facilitates efficient navigation and retrieval of specific files within the ZIP archive. Additionally, each entry in the ZIP file is preceded by a local header containing metadata about that particular entry, such as its filename, size, and compression method. These local headers provide detailed information about each entry and are positioned immediately before their corresponding data within the ZIP archive. This organizational structure allows for efficient access and manipulation of individual entries within the ZIP file, facilitating tasks such as extraction, modification, and analysis.

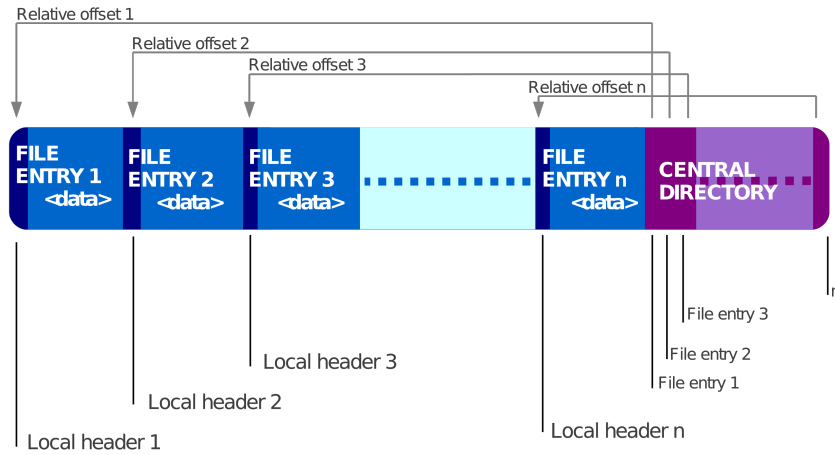


Figure 2.1: ZIP internal layout. Source: Wikipedia

2.1.1 General ZIP handling

Android relies on `libziparchive`¹ for most tasks involving decompression and file handling within the APK structure. Notably, the `ZipEntryCommon` structure² lacks the filename of each entry. Instead, Android memory-maps the zip central directory, loading a hash table with pointers to filenames. It should be noted that only filenames which are not null-terminated are included in the mapping³, while other fields are located at fixed offsets from the filename.

2.2 APK installation

The APK installation process on Android involves a coordinated effort between several components of the system, each playing a distinct role to ensure the successful installation and integration of the application.

When an APK is installed, it undergoes multiple stages of examination. Initially, a lightweight parsing operation is conducted using the `parseClusterPackageLite()` method⁴. This operation extracts essential metadata such as package name and version code from the APK, enabling preliminary validation steps.

¹<https://android.googlesource.com/platform/system/libziparchive>

²https://android.googlesource.com/platform/system/libziparchive/+/refs/heads/android14-release/include/ziparchive/zip_archive.h#45

³https://android.googlesource.com/platform/system/libziparchive/+/refs/heads/android14-release/zip_archive.cc#83

⁴<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/core/java/android/content/pm/PackageParser.java#987>

This involves extracting the `AndroidManifest.xml` file from the APK and parsing its contents⁵. The `AndroidManifest.xml` file serves as a blueprint for the Android system, containing vital information about the application’s structure, permissions, activities, and more. Extracting and parsing this file is critical for the Android system to acquire essential details about the package being installed.

In the context of our research, the extraction and parsing of `AndroidManifest.xml` is of significant importance. Therefore, identifying and understanding the exact points where these processes occur is essential for comprehending the intricate workings of the evasive tactics discovered in Android malware.

In the subsequent subsections, we elaborate on the extraction and parsing details of `AndroidManifest.xml`, offering insights into the procedures for both Android Oreo (version 8.1) and Android Pie (version 9). The inclusion of details regarding these two versions is motivated by a significant alteration in the extraction process of the `AndroidManifest.xml` between the releases.

Finally, an important thing that should be noted is that from Android version 9 onward, the extraction process remains consistent. For the latest versions of Android the `PackageParser.java` has been deprecated and the relevant functionality for the "lite" package parsing has been moved to `ApkLiteParseUtils.java`⁶. All methods mentioned in the next sections for API 28, do exist in the latest versions of Android in the `ApkLiteParseUtils.java` and have not changed.

2.2.1 Extraction of the `AndroidManifest.xml`

Android Oreo (API 27)

An initial lightweight parsing operation is conducted using the `AndroidManifest.xml` in order to fetch certain information about the application to be installed. This operation is starting from the method `parseApkLite`⁷ and in figure 2.2 the chain of method invocations can be seen, along with the corresponding paths and line numbers from the Android

⁵<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/core/java/android/content/pm/PackageParser.java#1507>

⁶<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/core/java/android/content/pm/parsing/ApkLiteParseUtils.java>

⁷<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/oreo-release/core/java/android/content/pm/PackageParser.java#1650>

source code.



Figure 2.2: AndroidManifest.xml extraction method invocation flow for Android Oreo

The final method responsible for the actual extraction of the AndroidManifest.xml is the `openAssetFromZipLocked`⁸. The purpose of this method is to create a new asset object based on an entry of a ZIP archive. The ZIP archive in this case is the APK to be installed and the `entryName` has the value of `AndroidManifest.xml`.

```

1  if (!pZipFile->getEntryInfo(entry, &method, &uncompressedLen, NULL, NULL,
2      NULL, NULL))
3  {
4      ALOGW("getEntryInfo failed\n");
5      return NULL;
6  }
7  FileMap* dataMap = pZipFile->createEntryFileMap(entry);
8  if (dataMap == NULL) {
9      ALOGW("create map from entry failed\n");
10     return NULL;
11 }
12 if (method == ZipFileRO::kCompressStored) {

```

⁸<https://android.googlesource.com/platform/frameworks/base/+refs/heads/oreo-release/libs/androidfw/AssetManager.cpp#857>

```

13     pAsset = Asset::createFromUncompressedMap(dataMap, mode);
14     ALOGV("Opened uncompressed entry %s in zip %s mode %d: %p", entryName.
15         string(),
16         dataMap->getFileName(), mode, pAsset);
17 } else {
18     pAsset = Asset::createFromCompressedMap(dataMap,
19         static_cast<size_t>(uncompressedLen), mode);
20     ALOGV("Opened compressed entry %s in zip %s mode %d: %p", entryName.
21         string(),
22         dataMap->getFileName(), mode, pAsset);
23 }

```

Listing 2.1: Excerpt from `openAssetFromZipLocked`

The snippet above shows an excerpt from the method `openAssetFromZipLocked`. In the first line, the method extracts the compression method and uncompressed length from the entry of the `AndroidManifest.xml` from the ZIP archive. In line 12 there is an if-statement checking if the compression method is that of `STORED`, where in this case it means that the data does not have any compression. In line 16 is the else-statement covering all other compression methods, which as we saw in Section 2.1, the only other expected compression method is that of `DEFLATED`, which means that data are compressed. Therefore, the code in line 17 calls the method `createFromCompressedMap` which creates the asset after decompressing the data first. This concludes the creation of the `AndroidManifest.xml` asset based on the compression method retrieved from the APK file.

Android Pie (API 28)

In the same way, as we saw in Section 2.2.1, the lightweight parsing of the APK, in Android Pie, starts also with the method `parseApkLite`⁹.

⁹<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/pie-release/core/java/android/content/pm/PackageParser.java#1551>

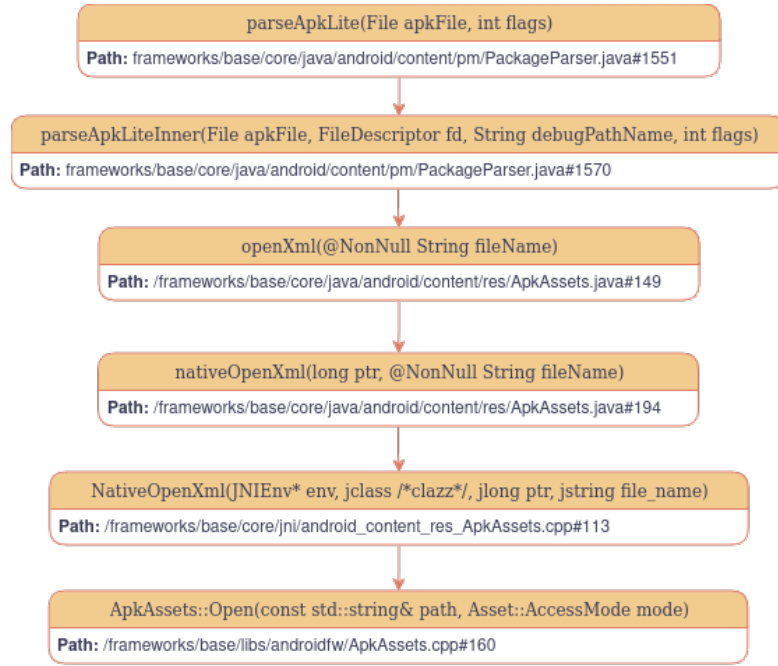


Figure 2.3: AndroidManifest.xml extraction method invocation flow for Android Pie

Figure 2.3 shows the chain of method invocations, with method `ApkAssets::Open` being the final one responsible for the extraction of the `AndroidManifest.xml`. The following excerpt from the `ApkAssets::Open`¹⁰ shows the actual source code responsible for the extraction.

```

1 if (entry.method == kCompressDeflated) {
2     std::unique_ptr<FileMap> map = util::make_unique<FileMap>();
3     if (!map->create(path_.c_str(), ::GetFileDescriptor(zip_handle_.get()), entry
4         .offset,
5         entry.compressed_length, true /*readOnly*/)) {
6         LOG(ERROR) << "Failed to mmap file '" << path << "' in APK '" << path_ << "
7         '"';
8         return {};
9     }
10    std::unique_ptr<Asset> asset =
11        Asset::createFromCompressedMap(std::move(map), entry.uncompressed_length,
12        mode);
13    if (asset == nullptr) {
14        LOG(ERROR) << "Failed to decompress '" << path << "'.";
15        return {};
16    }
17 }

```

¹⁰<https://android.googlesource.com/platform/frameworks/base/+refs/heads/pie-release/libs/androidfw/ApkAssets.cpp#160>

```

13 }
14 return asset;
15 } else {
16 std::unique_ptr<FileMap> map = util::make_unique<FileMap>();
17 if (!map->create(path_.c_str(), ::GetFileDescriptor(zip_handle_.get()), entry
    .offset,
18         entry.uncompressed_length, true /*readOnly*/) {
19     LOG(ERROR) << "Failed to mmap file '" << path << "' in APK '" << path_ << "
        '"';
20     return {};
21 }
22 std::unique_ptr<Asset> asset = Asset::createFromUncompressedMap(std::move(map
    ), mode);
23 if (asset == nullptr) {
24     LOG(ERROR) << "Failed to mmap file '" << path << "' in APK '" << path_ << "
        '"';
25     return {};
26 }
27 return asset;
28 }

```

Listing 2.2: Excerpt from ApkAssets::Open

In the first line the method of the ZIP entry, in this case the AndroidManifest.xml file, is being checked if it is DEFLATED, meaning that the data are compressed. If that is the case then in line nine the asset is being created from the method `createFromCompressedMap`. In line fifteen there is the else-statement, meaning that if the compression method has any other value besides eight, which is the value for the DEFLATED compression method, then the data of the asset are being treated as STORED. In line 22 the method `createFromUncompressedMap` is being called to create the asset from the uncompressed data.

It is worth highlighting that this behavior remains consistent across recent versions of Android. Even in the latest iterations, such as Android 14, while there may have been slight refinements and method refactoring, the extractions of the AndroidManifest.xml process remains unchanged. The final step is happening in method `ZipAssetsProvider::OpenInternal`¹¹, where you can see that the source code is similar to the previous snippet.

¹¹<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/AssetsProvider.cpp#156>

2.2.2 Parsing of the AndroidManifest.xml

The process of parsing the binary AXML format of the AndroidManifest.xml is overall the same for the different Android versions. A few changes have been introduced in the recent commits for Android version 14. As a reference the Android source code from `android14-release` will be used and whenever there are differences for the most recent versions, these will be pointed out.

The following paragraphs describe a few of the interesting points of the parsing process.

Type Identifier

The method `NativeOpenXml` that can be seen in figure 2.3, besides the extraction of the AXML binary, handles also the parsing of it¹². Method `ResXMLTree::setTo` is where the parsing starts¹³.

```
1 if (dtohs(mHeader->header.headerSize) > mSize || mSize > size) {  
2     ALOGW("Bad XML block: header size %d or total size %d is larger than data  
3         size %d\n",  
4             (int)dtohs(mHeader->header.headerSize),  
5             (int)dtohl(mHeader->header.size), (int)size);  
6     mError = BAD_TYPE;  
7     restart();  
8     return mError;  
9 }
```

Listing 2.3: Excerpt from `ResXMLTree::setTo`

The excerpt shown above, displays how the parsing of the `ResXMLTree_header` is occurring. It should be noted that there is no validation on whether the first two bytes of the incoming data are 03 00, which are the expected type identifier for the header chunk. The only validation that takes place is related to the size of the header.

In commit `a967411ae6ffb4172f9de0bfc19c586d46f5158b`¹⁴, thanks to Yurii Zubrytskyi, this changes. As can be seen in the following snippet as well, a check was added to verify

¹²https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/core/jni/android_content_res_ApkAssets.cpp#437

¹³<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#1770>

¹⁴<https://android.googlesource.com/platform/frameworks/base/+/a967411ae6ffb4172f9de0bfc19c586d46f5158b/libs/androidfw/ResourceTypes.cpp#1805>

that they type identifier of the header chunk is the expected one.

```
1 if (dtohs(mHeader->header.type) != RES_XML_TYPE) {
2     ALOGW("Bad XML block: expected root block type %d, got %d\n",
3         int(RES_XML_TYPE), int(dtohs(mHeader->header.type)));
4     mError = BAD_TYPE;
5     goto done;
6 }
```

Listing 2.4: Added check for the header type

The stringCount value

Continuing, the string pool is the next chunk to be parsed¹⁵. Method `ResStringPool::setTo` is responsible for this, and there are two interesting parts to be covered in it.

```
1 if (mHeader->stringCount > 0) {
2     if ((mHeader->stringCount*sizeof(uint32_t) < mHeader->stringCount) //
3         uint32 overflow?
4         || (mHeader->header.headerSize+(mHeader->stringCount*sizeof(uint32_t)
5         ))
6         > size) {
7         ALOGW("Bad string block: entry of %d items extends past data size %d\n",
8             (int)(mHeader->header.headerSize+(mHeader->stringCount*sizeof
9             (uint32_t))),
10             (int)size);
11         return (mError=BAD_TYPE);
12     }
```

Listing 2.5: Excerpt from `ResStringPool::setTo` checking `stringCount`

The first interesting aspect is related to the validation of the `stringCount` in line one of the presented snippet. This value, derived from the header of the string pool chunk, is expected to accurately represent the total number of strings within the string pool. Notably, the snippet includes checks to ensure the integrity and validity of the AXML file. Aside from verifying that the `stringCount` is positive, these checks guard against

¹⁵<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#1824>

potential overflow scenarios and validate that the calculated sizes do not surpass the actual size of the file.

The second interesting part is in the same method, in line 577¹⁶ of the source code of `ResourceTypes.cpp`, there is another check for the `styleCount` and based on that the actual string pool size is being calculated. The following snippet shows the relevant source code.

```
1 if (mHeader->styleCount == 0) {
2     mStringPoolSize = (mSize - mHeader->stringsStart) / charSize;
3 } else {
4     // check invariant: styles starts before end of data
5     if (mHeader->stylesStart >= (mSize - sizeof(uint16_t))) {
6         ALOGW("Bad style block: style block starts at %d past data size of %d\n",
7             (int)mHeader->stylesStart, (int)mHeader->header.size);
8         return (mError=BAD_TYPE);
9     }
10    // check invariant: styles follow the strings
11    if (mHeader->stylesStart <= mHeader->stringsStart) {
12        ALOGW("Bad style block: style block starts at %d, before strings at %d\n",
13            (int)mHeader->stylesStart, (int)mHeader->stringsStart);
14        return (mError=BAD_TYPE);
15    }
16    mStringPoolSize =
17        (mHeader->stylesStart - mHeader->stringsStart) / charSize;
18 }
```

Listing 2.6: Excerpt from `ResStringPool::setTo` calculating `mStringPoolSize`

The important thing to notice is that in both lines 2 and 16, the `mStringPoolSize` is being calculated based on the `stringsStart` header information and the `charSize`. The `stringCount` header is not used to calculate the `mStringPoolSize`.

¹⁶<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#577>

Strings in the string pool

In both UTF-8 and UTF-16 formats, the length of strings is indicated by a length encoded in the stored data, typically using 1 or 2 characters of length data. This encoding allows for a maximum length of 0x7FFF (32767 bytes) in UTF-8 and 0x7FFFFFFF (2147483647 bytes) in UTF-16¹⁷. However, exceeding these limits is discouraged due to potential abuse of string resources. To accommodate larger data sizes, an additional 2 bytes are provided if the high bit is set in the initial character of the length data, allowing for extended lengths to be encoded. In such cases, the high bit of the first character is dropped, and the values of both characters are combined to determine the actual length of the string. This mechanism ensures that strings with lengths beyond what can be represented by a single or two bytes can still be accommodated within the Android system.

Additionally, `ResStringPool::stringAt`¹⁸ includes more validation checks to verify that each string in the string pool is not malformed. These checks include a validation on the declared size that it does not exceed the string pool size and that the string is null-terminated.

Valid chunk header types

While parsing the data from the `AndroidManifest.xml`, there is a validation in each case for the header type of the next element. In method `ResXMLTree::setTo` the following snippet can be found.

```
1 while (((const uint8_t*)chunk) < (mDataEnd-sizeof(ResChunk_header))) &&
2       (((const uint8_t*)chunk) < (mDataEnd-dtohl(chunk->size)))) {
3     status_t err = validate_chunk(chunk, sizeof(ResChunk_header), mDataEnd, "
4     XML");
5     if (err != NO_ERROR) {
6         mError = err;
7         goto done;
8     }
9     [...]
```

¹⁷<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#714>

¹⁸<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#769>

```

9      if (type == RES_STRING_POOL_TYPE) {
10          mStrings.setTo(chunk, size);
11      } else if (type == RES_XML_RESOURCE_MAP_TYPE) {
12          mResIds = (const uint32_t*)
13              (((const uint8_t*)chunk)+dtohs(chunk->headerSize));
14          mNumResIds = (dtohl(chunk->size)-dtohs(chunk->headerSize))/sizeof(
uint32_t);
15      } else if (type >= RES_XML_FIRST_CHUNK_TYPE
16          && type <= RES_XML_LAST_CHUNK_TYPE) {
17          if (validateNode((const ResXMLTree_node*)chunk) != NO_ERROR) {
18              mError = BAD_TYPE;
19              goto done;
20          }
21          mCurNode = (const ResXMLTree_node*)lastChunk;
22          if (nextNode() == BAD_DOCUMENT) {
23              mError = BAD_TYPE;
24              goto done;
25          }
26          mRootNode = mCurNode;
27          mRootExt = mCurExt;
28          mRootCode = mEventCode;
29          break;
30      } else {
31          if (kDebugXMLNoisy) {
32              printf("Skipping unknown chunk!\n");
33          }
34      }
35      lastChunk = chunk;
36      chunk = (const ResChunk_header*)
37          (((const uint8_t*)chunk) + size);
38  }

```

Listing 2.7: Excerpt from ResXMLTree::setTo checking chunk header type

In line 3 there is a call to the method `validate_chunk`, this method ensures that the resource chunk conforms to expected size and alignment constraints. Lines 9, 11, and 15 include additional checks to ensure that the header type matches one of the expected values. If either the resource chunk validation or the header type check fails, the method will proceed to skip the current chunk and advance to the next one.

In line 17 the `validateNode` method and in line 22, the method `ResXMLParser::nextNode()`¹⁹ offer more fine grained validation for the exact element type expected. The following snippet shows the different cases the source code accounts for.

```
1 switch ((mEventCode=eventCode)) {
2     case RES_XML_START_NAMESPACE_TYPE:
3     case RES_XML_END_NAMESPACE_TYPE:
4         minExtSize = sizeof(ResXMLTree_namespaceExt);
5         break;
6     case RES_XML_START_ELEMENT_TYPE:
7         minExtSize = sizeof(ResXMLTree_attrExt);
8         break;
9     case RES_XML_END_ELEMENT_TYPE:
10        minExtSize = sizeof(ResXMLTree_endElementExt);
11        break;
12    case RES_XML_CDATA_TYPE:
13        minExtSize = sizeof(ResXMLTree_cdataExt);
14        break;
15    default:
16        ALOGW("Unknown XML block: header type %d in node at %d\n",
17              (int)dtohs(next->header.type),
18              (int)((((const uint8_t*)next)-((const uint8_t*)mTree.mHeader))));
19        continue;
20 }
```

Listing 2.8: Excerpt from `ResXMLParser::nextNode()`

It should be noted that when an unknown `eventCode` is encountered, the `ResXMLParser::nextNode()` method logs a warning and continues to the next XML node, bypassing further processing of the current node. This allows the XML parser to gracefully handle unknown XML block types while iterating through the XML data.

Parsing of element attributes

The elements comprising the `AndroidManifest.xml` can be identified through the identifier types of `RES_XML_START_ELEMENT_TYPE` for the start of the element and `RES_XML_END_ELEMENT_TYPE`

¹⁹<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#1647>

for the end of it. The `ResXMLTree::validateNode` method²⁰ is responsible for the first validation steps of all the chunks but more specifically for the `RES_XML_START_ELEMENT_TYPE` chunk. This method ensures that the attributes of an element are properly defined and within the bounds of the node.

Before moving further, we need to establish what are the properties of the attributes of an element. The following snippet shows how the `ResXMLTree_attribute` struct is defined²¹.

```
1 struct ResXMLTree_attribute
2 {
3     // Namespace of this attribute.
4     struct ResStringPool_ref ns;
5     // Name of this attribute.
6     struct ResStringPool_ref name;
7     // The original raw string value of this attribute.
8     struct ResStringPool_ref rawValue;
9     // Processed typed value of this attribute.
10    struct Res_value typedValue;
11};
```

Listing 2.9: Struct of `ResXMLTree_attribute`

The `ResStringPool_ref` structure contains an unsigned 32-bit integer (`uint32_t`), typically occupying 4 bytes of memory storage²². On the other hand, the `Res_value` struct contains an unsigned 16-bit integer (2 bytes), two unsigned 8-bit integers (1 byte each), and one unsigned 32-bit integer (4 bytes)²³. This breakdown allows us to pre-determine the memory required for the `ResStringPool_ref` struct. Summing up the byte sizes mentioned, we can ascertain that 20 bytes are necessary.

Additional information about the attributes of an element are provided by the extended XML tree node for start tags²⁴. This information includes the offset from the start of the structure where the attributes start and the size of the attribute structures as

²⁰<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#1883>

²¹<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#701>

²²<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#429>

²³<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#286>

²⁴<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#671>

well as the attributes count. Retrieving any of the properties of an attribute uses the information provided by the extended XML tree node for start tags. As an example, the snippet below shows the method that retrieves the namespace index for an attribute.

```
1 int32_t ResXMLParser::getAttributeNamespaceID(size_t idx) const
2 {
3     if (mEventCode == START_TAG) {
4         const ResXMLTree_attrExt* tag = (const ResXMLTree_attrExt*)mCurExt;
5         if (idx < dtohs(tag->attributeCount)) {
6             const ResXMLTree_attribute* attr = (const ResXMLTree_attribute*)
7                 (((const uint8_t*)tag)
8                 + dtohs(tag->attributeStart)
9                 + (dtohs(tag->attributeSize)*idx));
10            return dtohl(attr->ns.index);
11        }
12    }
13    return -2;
14 }
```

Listing 2.10: The ResXMLParser::getAttributeNamespaceID method

Notice how the `attributeStart` and the `attributeSize` are used to calculate the correct position on where the index should be found. This means that as long as these properties of the `ResXMLTree_attrExt` are correct, then the Android system would calculate the correct offset for the attribute properties. Do note that although the size an attribute occupies is theoretically predetermined, the `attributeSize` is still used to make sure the correct size is calculated. A similar approach is used for the rest of the properties of each attribute.

Attribute names and values

The following snippet shows method `ResXMLParser::getAttributeData`²⁵, which retrieves the value of an attribute from a start tag at a specified index, handling data of different types and performing value lookups if necessary, before returning the value to the caller.

```
1 int32_t ResXMLParser::getAttributeData(size_t idx) const
2 {
3     if (mEventCode == START_TAG) {
4         const ResXMLTree_attrExt* tag = (const ResXMLTree_attrExt*)mCurExt;
```

²⁵<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp#1482>

```

5         if (idx < dtohs(tag->attributeCount)) {
6             const ResXMLTree_attribute* attr = (const ResXMLTree_attribute*)
7                 (((const uint8_t*)tag)
8                 + dtohs(tag->attributeStart)
9                 + (dtohs(tag->attributeSize)*idx));
10            if (mTree.mDynamicRefTable == NULL ||
11                !mTree.mDynamicRefTable->requiresLookup(&attr->typedValue
12            )) {
13                return dtohl(attr->typedValue.data);
14            }
15            uint32_t data = dtohl(attr->typedValue.data);
16            if (mTree.mDynamicRefTable->lookupResourceId(&data) == NO_ERROR)
17            {
18                return data;
19            }
20        }
21        return 0;
22    }

```

Listing 2.11: The ResXMLParser::getAttributeData method

The interesting part to note, is in line 10, where if a value lookup is not necessary or a dynamic reference table is not available then the `attr -> typedValue.data` will be returned. This means that if for example an unexpected value is stored in the `typedValue.data`, then this will be returned.

Extended XML tree node for namespace end nodes

The struct for the extended XML tree node for namespace start and end nodes is shown in the following snippet as it is defined in the `ResourceTypes.h` header file²⁶

```

1 struct ResXMLTree_namespaceExt
2 {
3     // The prefix of the namespace.
4     struct ResStringPool_ref prefix;
5
6     // The URI of the namespace.

```

²⁶<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#643>

```
7     struct ResStringPool_ref uri;  
8 };
```

Listing 2.12: Struct for the extended XML tree node for namespace start/end nodes.

The `ResStringPool_ref` structure, as we saw earlier as well, contains an unsigned 32-bit integer. This indicates that the size required by the `ResXMLTree_namespaceExt` can be described as twice the size of an unsigned 32-bit integer, with each integer typically requiring 4 bytes of memory storage. For this reason the minimum size set for the namespace end node as calculated in the `ResXMLParser::nextNode()`²⁷ is anticipated to be of 8 bytes in total.

²⁷<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/ResourceTypes.cpp/#1679>

3. Problem Definition

Recent years have seen a surge in app signing vulnerabilities[6], such as Janus (CVE-2017-13156) and Master Key (CVE-2013-4787), exposing Android apps and devices to significant security risks. These vulnerabilities allowed attackers to manipulate APKs without invalidating their signatures, exploiting parsing anomalies in the APK's ZIP structure. Although in our case study the impact is far less devastating, it still highlights that parsing anomalies in the APK's ZIP and AXML structure, which can be leveraged to hide or delay uncovering malicious activities.

In 2023, there were over 3000 reported cases of malware utilizing evasion techniques to evade detection from static analysis[7]. Google's awareness of this issue dates back to at least May 9, 2023, when researchers from ThreatFabric first reported malware using similar evasion tactics[2]. The fact that Google recognized this by awarding a bug bounty to the researchers underscores the significance of our study, highlighting the potential impact of delayed detection of malicious intents. Furthermore, Google acknowledged that some of its developer tools, such as APK Analyzer, currently struggle to parse such malicious applications, treating them as invalid while still allowing them to be installed on user devices. Recent reports as of April 2024 indicate the continued emergence of known malware families employing similar evasion techniques[5].

In the upcoming sections, we will dive deeper into the insights provided in Chapter 2. By analyzing a series of recent Android malware instances that utilize evasive tactics leveraging the anomalies in the APK's ZIP and AXML structure, we will illustrate the creative ways we discovered in which malware exploits edge cases to evade static analysis.

3.1 Exploiting the ZIP structure

3.1.1 Tampered compression methods

In section 2.1, we discussed the `STORED` and `DEFLATED` compression methods utilized in the Android environment. Furthermore, in section 2.2.1, we explored the extraction process

for the AndroidManifest.xml entry from an APK, contrasting the procedures for Android Oreo and earlier versions with those for Android Pie and subsequent versions. Regardless of the Android version, the extraction of the AndroidManifest.xml follows a common approach: Android’s source code first checks whether the compression method matches one expectation, and if not, it assumes the other method. Since compression methods are represented by integer numbers, this essentially means that Android checks if the expected type corresponds to one number, and any deviation prompts the assumption of the opposite type. This reliance on assumptions is exploited by malware, as other tools handling zip files typically explicitly verify the compression method without making such assumptions. Consequently, if a random number not corresponding to a compression method is designated for the compression method of the AndroidManifest.xml, Android’s source code would capture it in the “else-statement,” while existing zip file handling tools would throw an exception as they fail to recognize the compression method.

Using as an example the source code from Android Pie(API 28), as shown in snippet 2.2.1, we observe that when the AndroidManifest.xml is added as an entry in an APK with a genuine compression method of `STORED`, but with a compression method value that diverges from 0 or 8, Android will interpret it as `STORED`, regardless of the compression method detected. Figure 3.1 is depicting the two scenarios for the different Android versions and when common ZIP tools and libraries fail to process the APK.

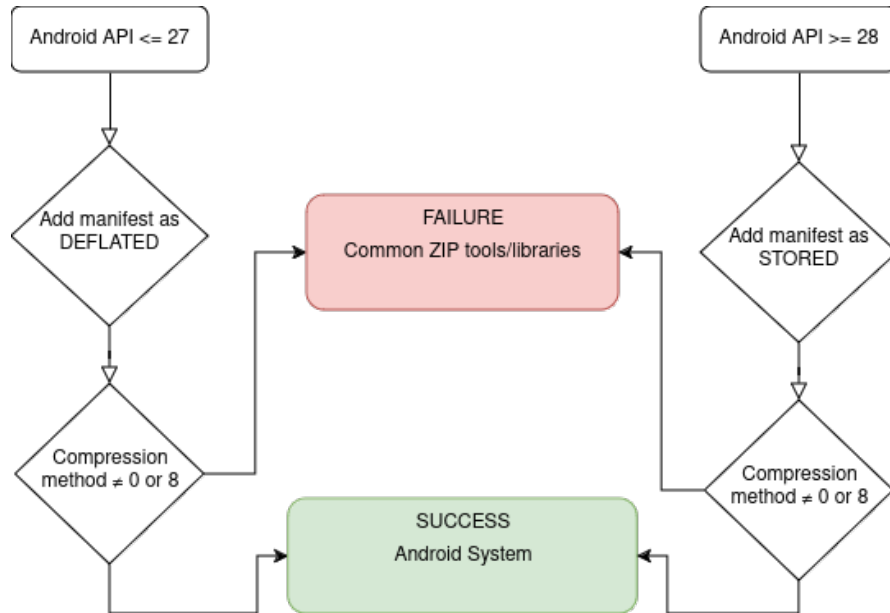


Figure 3.1: AndroidManifest.xml extraction failure or success

Showcase example

A recent malware from February 2024¹ was picked to illustrate this evasion tactic.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00:0000	50	4B	03	04	0A	00	00	08	C7	0F	D8	B2	99	57	0D	AE	PK.....C.0 ² ™W.®
00:0010	A4	DF	7B	17	00	00	18	43	00	00	13	00	03	00	41	6E	®{....C.....An
00:0020	64	72	6F	69	64	4D	61	6E	69	66	65	73	74	2E	78	6D	droidManifest.xml
00:0030	6C	00	00	00	00	00	08	00	18	43	00	00	01	00	1C	00	l.....C.....
00:0040	88	19	00	00	AC	00	00	00	00	00	00	00	00	00	00	00	^.....

Figure 3.2: The local header entry of the AndroidManifest.xml from the APK

Figure 3.2 shows the bytes of the local header for the AndroidManifest.xml entry. The offset for the compression method in the local header is 8 bytes[1] and it takes up 2 bytes. These are the highlighted 2 bytes shown in figure 3.2. The integer value it translates is 4039, which does not correspond to any of the existing compression methods. Notice also that the compressed size and the uncompressed size of the entry which are located at offsets 18 and 22 respectively and occupy 4 bytes, have the values of 6011 for the compressed size and 17176 for the uncompressed.

For comparison the central directory header is provided as well in figure 3.3.

2D:4FF0	41	50	4B	20	53	69	67	20	42	6C	6E	63	6B	20	34	32	APK Sig Block 42
2D:5000	50	4B	01	02	14	00	0A	00	00	08	A0	58	D8	B2	99	57	PK.....X0 ² ™W
2D:5010	0D	AE	A4	DF	7B	17	00	00	18	43	00	00	13	00	00	00	®{....C.....
2D:5020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	41	6EAn
2D:5030	64	72	6F	69	64	4D	61	6E	69	66	65	73	74	2E	78	6D	droidManifest.xml
2D:5040	6C	50	4B	01	02	14	00	0A	00	00	08	00	00	D8	B2	99	lPK.....0 ² ™
2D:5050	57	35	DC	DF	46	84	39	00	00	84	39	00	00	0E	00	00	WSÜBF„9.....
2D:5060	00	00	00	00	00	00	00	00	00	00	00	4C	43	00	00	72	LC r

Figure 3.3: The central directory entry of the AndroidManifest.xml from the APK

The values of the compression method and the compressed and uncompressed size occupy the same amount of bytes as in the local header, but have different offsets. The offsets are 10, 20 and 24 for the compression method and the compressed and uncompressed size respectively and are highlighted in figure 3.3 in the second box. The values translated to integers are the same for the compressed and uncompressed sizes of the local header we saw earlier, though the compression method has a different value (bytes A0 58) of 22688.

¹<https://bazaar.abuse.ch/sample/100f43e08c8fdca84e6c6759d254305bc16e657bed3e1d532d462341dfe07cc8/>

As explained extensively in this section but also in section 2.2.1, the malware should be using the `STORED` compression method in order for it to be installable in newer Android versions. So regardless of the different compression methods stated in the local header and in the central directory, the actual compression method is `STORED`.

3.1.2 Zipentry with empty filename

In section 2.1.1, it was highlighted that Android disregards filenames that are null-terminated when mapping zip entries. This implies that attackers could potentially insert an entry in an APK with an empty or null-terminated filename, along with properties such as compressed size and filename length that could disrupt standard zip handling tools. While Android can properly parse and discard such entries, other zip tools may encounter issues when attempting to process them. The following example showcases this attempt by a real malware sample.

Showcase example

The malware sample we examine in this case, has emerged in 2024 and has an MD5 hash of `fa8b1592c9cda268d8affb6bceb7a120`².

56:1000	50 4B 01 02	14 00 14 00	00 08 08 00	7B 7B 78 57	PK.....{xw
56:1010	B8 73 D3 D3	59 00 00 00	54 00 00 00	00 00 00 00	,s00Y...T.....
56:1020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 50 4BPK
56:1030	01 02 14 00	0A 00 00 08	00 00 EB 8A	78 57 07 4CëŠxw.L

Figure 3.4: The central directory entry with an empty filename

Figure 3.4 shows a central directory entry, where there are three highlighted chunks. The first chunk of 4 bytes is the compressed size and has a value that translates to 89, the second chunk is the uncompressed size with a value of 84 and the last chunk of 2 bytes is the filename length. As you can see the filename length is declared to be empty and therefore it will be skipped by the Android system.

When attempting to read the local header of this entry though, we find different values.

²<https://tria.ge/240418-h2h1haed74>

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00:0000	50	4B	03	04	DD	DE	FF	FF	FF	FF	FF	DD	DE	FF	ED	CC	PK..ŸpīīīīīŸpŷīī
00:0010	DD	DE	ED	CD	DE	EE	ED	DD	DD	EE	EE	DD	DD	DD	01	54	ŸpīīpīīŸŷīīŸŷŷ.T
00:0020	00	AB	FF	06	7C	05	20	A1	BE	E0	09	04	0C	9F	98	79	.«ŷ. . i¼à...Ÿ~y
00:0030	B7	6B	29	FB	AB	41	38	77	E5	2C	89	64	3F	EA	78	00	·k)Ű«A8wâ,¼d?êx.
00:0040	B6	05	3C	80	82	82	14	A1	73	07	87	35	3E	87	C3	66	¶.<€,,.is.‡5>‡Ãf

Figure 3.5: The local header of the entry with an empty filename

The section highlighted in red in Figure 3.5 displays the first 4 bytes representing the compressed size, valued at 4007579117, followed by the uncompressed size at 4007517677, and the final 2 bytes representing the filename length, equal to 56814. These manipulated values are evidently intended to cause parsing errors in tooling designed to process the local headers of the APK.

3.2 Exploiting the AXML structure

In section 2.2.2, we presented in detail certain interesting aspects of the AndroidManifest.xml parsing process. Building upon that, this section adopts a similar approach to present how the aforementioned points in the parsing process can be exploited to break existing tooling and evade detection.

3.2.1 Spoofing the Type Identifier

In section 2.2.2, we examined the relevant Android source code responsible for processing the initial header of the AndroidManifest.xml. It was noted that prior to commit [a967411ae6ffb4172f9de0bfc19c586d46f5158b](#), there was no explicit validation of the first two bytes of the header; instead, the source code proceeded with processing. However, with the introduction of the mentioned commit, a validation step for the type identifier was added.

The absence of validation for the type identifier until the aforementioned commit implies that the parsing process remained unaffected by unexpected bytes in the first two bytes of the AndroidManifest.xml file. This, however, differs from other tools designed to handle AXML documents, which typically incorporate a validation step for the type identifier by default. Consequently, tampering with the first two bytes of the AndroidManifest.xml could potentially disrupt certain tools used for decoding AXML files, preventing them

from completing the conversion process.

Showcase example

Using again the same malware as in the previous example, we can see in figure 3.6 the highlighted first two bytes being 00 00. As we already discussed in section 2.2.2, the expected type identifier is 03 00.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000	00	00	08	00	18	43	00	00	01	00	1C	00	88	19	00	00C.....^...
0010	AC	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00
0020	00	00	00	00	00	00	00	00	04	00	00	00	08	00	00	00
0030	22	00	00	00	26	00	00	00	2A	00	00	00	2E	00	00	00	"...&...*.....

Figure 3.6: The spoofed type identifier of the header of the AndroidManifest.xml

3.2.2 Tampered stringCount value

In section 2.2.2, we explore the significance of the `stringCount` value in determining the size of the string pool. While `stringCount` theoretically denotes the expected number of strings within the pool, it's noteworthy that the Android source does not directly utilize this value; rather, it calculates the size. However, tools tasked with converting the AndroidManifest.xml into a human-readable format may rely on and trust the `stringCount` value. Consequently, tampering with `stringCount` could lead to inaccurate parsing of the manifest, likely resulting in errors.

This scenario is precisely what malware authors exploit by modifying the `stringCount` value to an incorrect representation of the stored strings, they can cause errors in tools that trust `stringCount` without recalculating the string pool size.

Showcase example

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000	03	00	08	00	18	43	00	00	01	00	1C	00	88	19	00	00C.....^...
0010	AC	00	00	00	00	00	00	00	00	00	00	00	00	02	00	00
0020	00	00	00	00	00	00	00	00	04	00	00	00	08	00	00	00
0030	22	00	00	00	26	00	00	00	2A	00	00	00	2E	00	00	00	"...&...*.....
0040	2A	00	00	00	2E	00	00	00	64	00	00	00	70	00	00	00

Figure 3.7: The string pool header

Figure 3.7 has highlighted the `ResStringPool_header`³ of the string pool. Right after the first 8 bytes which are the `ResChunk_header` is the `stringCount` which has the value of `AC 00 00 00`, which translates to the integer 172. The `stringsStart` is a few bytes further with value `00 02 00 00`, which translates to 512. The `stringsStart` is the index from header of the string data, which means it skips only the initial header of 8 bytes. Therefore, the strings should start at offset 520 and the initial `ResChunk_header` along with the `ResStringPool_header` occupy 36 bytes. This means that there are 484^4 remaining bytes in between for the string offsets and knowing that each string offset is 4 bytes long, this means that there are 121^5 true strings. Therefore the `stringCount` is tampered.

3.2.3 Strings surpassing maximum length

In Section 2.2.2, we discussed how the Android system allows for adding strings to the string pool that exceed typical size limits. Moreover, we explored the validation procedures implemented during string parsing to ensure that restrictions to size and structure are applied. However, tools that lack similar validation processes to those of the Android system might encounter errors when attempting to parse strings. These errors can arise due to various factors like length values that exceed even the size of the APK. The following example shows such a case.

Showcase example

A malware sample that we saw earlier as well is used again⁶.

0:0320	73 00 00 00	98 AB 4C 03	35 03 58 03	07 03 39 03	s...L.5.X...9.
0:0330	44 03 14 03	0E 03 4E 03	46 03 21 03	42 03 47 03	D....N.F.!B.G.
0:0340	2B 03 2B 03	25 03 53 03	04 03 52 03	54 03 20 00	+.%S...R.T. .
0:0350	38 03 61 00	50 03 1C 03	58 03 1E 03	64 03 45 03	8.a.P...X...d.E.
0:0360	0A 03 20 03	49 03 2B 03	58 03 5A 03	25 03 53 03	..I.+X.Z.%S.
0:0370	23 03 2A 03	6A 00 0F 03	4B 03 19 03	49 03 19 03	#,*.j...K...I...

Figure 3.8: Excessively long entry in string pool

Figure 3.8 highlights two sets of bytes. The first is the declared length of this string entry in the string pool, which has the value of 43928 and the second is the added bytes

³<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#460>

⁴520-36

⁵484/4

⁶<https://tria.ge/240418-h2h1haed74>

to "fix" the length. Following the logic explained in Section 2.2.2 and knowing that these are UTF16 encoded, the updated length is 731382604 which surpasses by a lot the size of the string pool. Therefore, this string is by default skipped by the Android system.

3.2.4 Invalid data between elements

The elements within the AndroidManifest.xml typically do not contain any additional data between them. However, as discussed in Section 2.2.2, even if they did, the Android system would not encounter issues, as it can simply discard any irrelevant data that does not match the expected chunk header types. Android validates whether the current chunk aligns with the expected headers and proceeds accordingly. While this validation step is not always necessary for most AndroidManifest.xml files, where elements are typically sequential, it exposes a vulnerability for AXML-handling tools that assume sequential elements. Exploiting this, malware authors could insert dummy data between elements, with the only restriction being to avoid the set of bytes that would mark a valid chunk header type.

Showcase example

1B20	00 00 00 00	02 01 10 00	80 00 00 00	00 00 00 00
1B30	FF FF FF FF	FF FF FF FF	27 00 00 00	14 00 18 00
1B40	03 00 00 00	00 00 00 00	FF FF FF FF	04 00 00 00
1B50	FF FF FF FF	08 00 00 10	18 00 00 00	00 00 00 00
1B60	FF FF FF FF	05 00 00 00	FF FF FF FF	08 00 00 10
1B70	1C 00 00 00	00 00 00 00	1E 00 00 00	78 00 00 00
1B80	7F 0C 00 00	08 00 00 03	7F 0C 00 00	00 00 00 00
1B90	84 80 01 00	00 00 00 00	00 00 00 00	00 00 00 00
1BA0	00 00 00 00	<u>03 01</u> 10 00	18 00 00 00	00 00 00 00
1BB0	FF FF FF FF	FF FF FF FF	27 00 00 00	02 01 10 00
1BC0	6C 00 00 00	02 00 00 00	FF FF FF FF	FF FF FF FF

Figure 3.9: Dummy data between elements of the AndroidManifest.xml

Figure 3.9 shows highlighted, with dark background, the bytes belonging to one element and underlined are the type identifier bytes of the next element. Do notice that there

are several bytes in between them that serve no functional purpose other than thwarting static analysis tools that do not follow a proper way to parse the file.

3.2.5 Unexpected attribute size

As discussed in detail in Section 2.2.2, despite the attributes' size being pre-determined due to the fixed-size members within the structures, Android developers opted to calculate it dynamically in each instance. This approach, utilizing information from the extended XML tree node for start tags (`ResXMLTree_attrExt`), offers a more robust solution. However, certain AXML parsing tools may deviate from this methodology. Consequently, if an equivalent amount of dummy data were appended to the end of the attributes and the `attributeSize` of the `ResXMLTree_attrExt` adjusted accordingly, Android would still parse it accurately. However, tools assuming attributes of a specific size would fail to handle this scenario appropriately.

Showcase example

For this use case the malicious APK with MD5 hash `f0d49de9f2e110bc989249c0deb7f8af`⁷ was used. Figure 3.10 shows in the first red square the bytes `18 00` which represent the `attributeSize`⁸ and translates to the integer value of 24. As already discussed, the pre-determined size of the `ResXMLTree_attribute` is 20 bytes, while the declared size is 24. This means that there are 4 additional dummy bytes added. These are the `00 00 00 00` and can be seen in figure 3.10 between the first attribute highlighted with dark background and the second red rectangle showing the first bytes of the next attribute.

24D0	FF FF FF FF	FF FF FF FF	95 00 00 00	02 01 10 00	yyyyyyyy•.
24E0	68 00 00 00	0C 00 00 00	FF FF FF FF	FF FF FF FF	h.....yyyyyyyy
24F0	94 00 00 00	14 00 18 00	02 00 00 00	00 00 00 00	".....
2500	FF FF FF FF	24 00 00 00	58 00 00 00	08 00 00 03	yyyy\$.X.....
2510	58 00 00 00	00 00 00 00	81 00 00 00	96 00 00 00	x.....
2520	72 2E 00 00	08 00 00 03	72 2E 00 00	00 00 00 00	r.....r.....
2530	76 DF 00 00	00 00 00 00	00 00 00 00	00 00 00 00	vß.....
2540	00 00 00 00	03 01 10 00	18 00 00 00	0C 00 00 00
2550	FF FF FF FF	FF FF FF FF	94 00 00 00	02 01 10 00	yyyyyyyy".

Figure 3.10: Dummy data between attributes of an element

⁷<https://tria.ge/230909-dclymsha73>

⁸<https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android14-release/libs/androidfw/include/androidfw/ResourceTypes.h#684>

3.2.6 Unexpected attribute names or values

In Android, each element can contain a set of attributes that define various aspects of its behavior. For instance, the `manifest` element, serving as the root element in the `AndroidManifest.xml` file, requires certain attributes while others are optional. The official Android developer guide outlines a predefined set of attributes applicable to it⁹. However, it is important to note that this list may not be exhaustive, and additional attributes may be permitted.

As discussed in Section 2.2.2, the value of an attribute can be retrieved from the `typedValue.data` field, which can hold arbitrary values constrained only by the `typedValue.dataType` associated with it.

This flexibility is precisely what Android malware authors exploit to disrupt the functionality of tools used for reverse engineering APKs. When tools are expecting a specific set of attributes for each element, receiving an unexpected one may cause disruptions. For instance, an attribute like `tag`¹⁰ may be introduced to the `manifest` element in an attempt to disrupt the normal flow of the the tool analyzing that malware. Our analysis of malware instances reveals such attempts and the following showcase is one example using the `manifest` element.

Showcase example

```
2340 02 01 10 00 FC 00 00 00 02 00 00 00 FF FF FF FF ....ü.....yyyý
2350 FF FF FF FF 84 00 00 00 14 00 18 00 08 00 00 00 yyyý.....
2360 00 00 00 00 81 00 00 00 14 00 00 00 FF FF FF FF .....yyyý
2370 08 00 00 10 2B 00 00 00 00 00 00 00 FF FF FF FF ....+.....yyyý
2380 15 00 00 00 26 00 00 00 08 00 00 03 26 00 00 00 ....&.....&...
2390 00 00 00 00 FF FF FF FF 20 00 00 00 FF FF FF FF ....yyyý ...yyyý
23A0 08 00 00 10 21 00 00 00 00 00 00 00 FF FF FF FF ....!.....yyyý
23B0 21 00 00 00 25 00 00 00 08 00 00 03 25 00 00 00 !...%.....%...
23C0 00 00 00 00 FF FF FF FF 8B 00 00 00 86 00 00 00 ....yyyý<...t...
23D0 08 00 00 03 86 00 00 00 00 00 00 00 FF FF FF FF ....t.....yyyý
23E0 8D 00 00 00 FF FF FF FF 08 00 00 10 21 00 00 00 ....yyyý....!...
23F0 00 00 00 00 FF FF FF FF 8E 00 00 00 FF FF FF FF ....yyyýŽ...yyyý
2400 08 00 00 10 0D 00 00 00 00 00 00 00 81 00 00 00 .....
2410 96 00 00 00 62 30 00 00 08 00 00 03 62 30 00 00 -...b0.....b0..
2420 00 00 00 00 6E 46 00 00 00 00 00 00 00 00 00 00 ....nF.....
2430 00 00 00 00 00 00 00 00 00 00 00 00 02 01 10 00 .....
2440 88 00 00 00 07 00 00 00 FF FF FF FF FF FF FF FF ^.....00000000
```

Figure 3.11: Unexpected attributes in manifest element

⁹<https://developer.android.com/guide/topics/manifest/manifest-element>

¹⁰<https://android.googlesource.com/platform/frameworks/base/+refs/heads/android14-release/core/res/res/values/public-final.xml#237>

Figure 3.11 highlights three key areas within the analyzed data. The first rectangle, containing bytes 84 00 00 00, corresponds to the integer 132, representing the index from the string pool resolving to the value "manifest". This shows that the current element is the manifest element, which, as discussed earlier, typically includes specific expected attributes. The second highlighted box denotes the last attribute of the element, indicated by bytes 96 00 00 00, translating to the integer 150. Again, this references an index from the string pool, resolving to the value "tag". The byte 03 signifies that the data should be interpreted as a string, with the actual data represented by bytes 62 30 00 00, corresponding to the integer 12386. Here, we observe the unexpected attribute "tag" added to the manifest element, accompanied by a placeholder value.

3.2.7 Zero size header for namespace end nodes

Malware may manipulate the `ResChunk_header` structure, particularly the field `uint32_t size;`, which denotes the total size of the chunk in bytes. These headers are found at the beginning of every data chunk in an Android resource. By setting the `size` field to 0, certain parsing tools that rely on reading this header to interpret the rest of the data fail to do so. However, as we saw in section 2.2.2, Android itself already sets the minimum size for the namespace end node and therefore is not affected.

Showcase example

For this use case the malware with MD5 hash 8151be0db0610bcab0eda728158287cd¹¹ was used. Figure 3.12 shows the namespace end element.

```

9560 9E 00 00 00 FF FF FF FF FF FF FF FF 58 00 00 00 z...yyyyyyxx...
9570 03 01 10 00 18 00 00 00 9F 00 00 00 FF FF FF FF .....Y...yyy
9580 FF FF FF FF 4D 00 00 00 03 01 10 00 18 00 00 00 yyyM.....
9590 A0 00 00 00 FF FF FF FF FF FF FF FF 29 00 00 00 ...yyy)...)
95A0 01 01 10 00 00 00 00 00 00 00 00 00 00 00 00 .....
95B0 23 00 00 00 24 00 00 00 #...$...

```

Figure 3.12: Namespace end element with zero size header

The highlighted bytes 00 00 00 00 represent the `size` field of the `ResChunk_header`. This essentially means that the declared size of the chunk is zero, which is clearly not true as the header alone is 16 bytes and then the `ResXMLTree_namespaceExt` is 8 more bytes.

¹¹<https://tria.ge/240311-b3mlraha95>

4. Approach

In this chapter, we explain the methodology for our empirical validation of the efficiency of current static analysis tools against sophisticated malware employing evasive techniques. Our research highlights significant gaps in the capabilities of conventional tools to detect and analyze malware employing evasive techniques such as the ones illustrated in Chapter 3. Building on the foundational insights presented, we introduce a novel tool specifically designed to bridge this gap. The upcoming sections will outline a structured approach for testing both existing and our newly developed tool across three distinct scenarios, each designed to simulate real-world conditions of malware evasion.

4.1 Collection of APKs

A social media post by JoeSecurity initially sparked this research, leading us to utilize the JoeSandbox platform¹ as a primary source to collect examples of malware employing evasive techniques during the starting stage. Additional online analysis providers, including Triage² and MalwareBazaar³, were also instrumental in identifying such malware. Our methodology involved not only downloading all available APK files from these providers but also tracking uploaded APKs that triggered specific errors. For instance, both JoeSandbox and Triage allowed us to pinpoint APKs employing evasive methods based on error notifications displayed on their respective interfaces.

Rule	Description	Author
JoeSecurity_apk_invalid_zip_compression	Yara detected apk with invalid zip compression	Joe Security

Figure 4.1: JoeSandbox YARA rule identifying ZIP structure evasion detected.

¹<https://www.joesandbox.com/>

²<https://tria.ge/>

³<https://bazaar.abuse.ch>

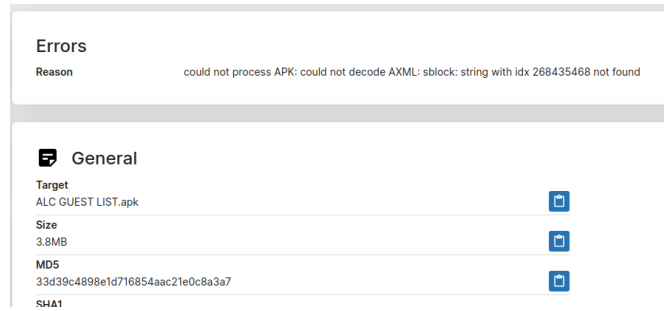


Figure 4.2: Error appearing in the static analysis tab of Triage

It is important to note the observed trend in the frequency of these errors. An analysis of Triage data from August 2023 revealed only a few instances, whereas by April 2024, there were consistently at least ten cases daily. This significant increase corroborates the hypothesis that evasion techniques are being increasingly sold as a service by underground platforms[2].

Throughout the development of apkInspector since September 2023, although an exact tally was not maintained, over 100 different APKs were utilized to test and validate the effectiveness of the tool. This extensive testing ensured the reliability and accuracy of the results in detecting and analyzing evasive malware.

4.2 Current Tooling Landscape

The selected tools for testing the evasive techniques used by malware include some of the most widely used and renowned tools in the field of Android reverse engineering.

4.2.1 Jadx

Jadx⁴ is a decompiler for Android applications that converts APK files into source code directly. This tool is especially valuable for developers interested in examining the code of Android apps to understand their functionality or identify any malicious code segments. It provides a GUI (Graphical User Interface) for easier navigation and analysis. Jadx is primarily written in Java, which enhances its compatibility and performance across different platforms.

⁴<https://github.com/skylot/jadx>

4.2.2 Apktool

Apktool⁵ is a tool for reverse engineering Android apk files. It allows you to decode resources to nearly original form and rebuild them after making modifications. It is highly useful for understanding the underlying architecture and design of an app. Apktool is written in Java, making it accessible across various operating systems with Java support.

4.2.3 ApkAnalyzer

ApkAnalyzer⁶ is a tool developed by Google that provides insights into the composition of APK files, helping developers to reduce the size of their applications and improve performance. It helps in identifying unnecessary resources and inspecting the manifest file and permissions. ApkAnalyzer is integrated into Android Studio, which is based on the IntelliJ platform primarily using Java. This tool assists developers in optimizing and understanding their Android application packages effectively.

4.2.4 Androguard

Androguard⁷ is a comprehensive suite of tools designed for Android APK reverse engineering and analysis. It can decode, analyze, and create visual representations of the Android application, which includes the source code, xml files and manifest. This tool facilitates the understanding of the overall structure and flow of an application. Androguard is written in Python, offering a flexible and scriptable environment that integrates well with other forensic and analysis tools, such as Mobile Security Framework (MobSF)⁸ and Viper⁹.

4.3 Empirical Testing Methodology

Our testing methodology is split into three distinct stages, each designed to incrementally address and test the resilience of static analysis tools against various evasive techniques used by malware.

⁵<https://github.com/iBotPeaches/Apktool/>

⁶<https://developer.android.com/tools/apkanalyzer>

⁷<https://github.com/androguard/androguard/>

⁸<https://github.com/MobSF/Mobile-Security-Framework-MobSF>

⁹<https://github.com/viper-framework/viper>

Stage One: We begin with the original version of the Android malware APK that initiated this research¹⁰. Using the selected tools, we will attempt to analyze this APK in its original state. As detailed in Section 3.1, we anticipate that the tools will fail to process the APK due to the evasion techniques related to the ZIP structure that it employs.

Stage Two: In this stage, we modify the malware by correcting the evasion techniques associated with the ZIP structure, while retaining those related to the Android XML (AXML). Despite these modifications, we expect that the tools will still struggle to successfully analyze the APK. However, as of April 2024, some tools may have incorporated updates that address the type identifier validation issues discussed in Section 2.2.2. Thus, while failures are expected, they may occur due to different issues related to the AXML structure.

Stage Three: We resolve the evasive tactics associated with the type identifier, as explored in Section 3.2.1. This adjustment gives tools that previously failed in the second stage another opportunity to process the APK, focusing on any remaining evasion techniques.

The objective of this structured approach is to demonstrate that even with partial mitigation of the evasive techniques, our most sophisticated and commonly used tools still face significant challenges in effectively parsing and analyzing the APK. This highlights the critical need for advancements in our tooling to keep pace with evolving malware strategies.

4.4 Development of the New Tool

apkInspector¹¹ was developed with a clear focus on addressing the evasion techniques that have challenged existing Android static analysis tools. Since its inception in September 2023, the tool has been tailored to tackle the issues identified in malware analyzed from August 2023 onward. Its design is inherently modular, separating the handling of ZIP-related evasion techniques from those involving Android XML (AXML) files. This separation enhances the tool’s adaptability and effectiveness in focusing on specific

¹⁰<https://tria.ge/240425-kfqjkhb5x/static1>

¹¹<https://github.com/erev0s/apkInspector>

evasion tactics.

The core innovation of apkInspector lies in its robust handling of ZIP and AXML evasion techniques. The ZIP module, in particular, allows decompressing entries from APKs, which not only aids in malware analysis but also positions apkInspector as a potential substitute for other decompression libraries within various tools. This capability ensures that apkInspector can be seamlessly integrated into broader security and analysis workflows, enhancing its utility and reach.

apkInspector is designed to function both as a Command Line Interface (CLI) and as a library within Python projects, offering flexibility in deployment and integration.

The CLI aspect of apkInspector allows users to perform detailed analysis through straightforward commands, making it accessible for both novice users and expert analysts. As a Python library, apkInspector provides significant extensibility. It can be integrated into larger Python applications or used in conjunction with other tools, such as Androguard and Medusa¹², with which it is already integrated. This integration highlights the tool’s reliability and the trust it has gained within the developer community.

apkInspector has rapidly established itself as a trusted tool within the Android security community. Beyond its rigorous validation against over 1800 apps from the Google Play Store¹³, which confirmed the high quality of its analytical results, the tool has also been integrated into well-regarded community projects like Androguard and Medusa. This integration, ongoing for several months, not only highlights apkInspector’s reliability and effectiveness but also underscores its critical contribution to advancing the field of Android malware analysis. The extensive testing and community adoption collectively confirm the pivotal role of apkInspector in enhancing the capabilities of malware detection and analysis.

4.4.1 Empirical Testing Methodology

In Chapter 5, we will detail the performance of apkInspector through two focused sets of tests. First, we will subject apkInspector to the same three-stage testing process

¹²<https://github.com/Ch0pin/medusa>

¹³https://github.com/erev0s/apkInspector/tree/main/tests/top_apps

applied to other tools, designed to evaluate its resilience against sophisticated evasion techniques. Following this, we will present a test performed to assess the accuracy of apkInspector in handling standard, non-malicious applications from the Play Store, comparing its outputs with those of established tools to ensure its reliability in everyday scenarios.

4.4.2 Integration with Existing Tooling

It is important to first note that apkInspector has already been successfully integrated into both Androguard and Medusa as mentioned above. Specifically for Androguard, apkInspector was officially incorporated on October 23, 2023, as evidenced by the commit history of the project. At that time, we were not maintainers of the Androguard project; our role as maintainers was granted in December 2023, subsequent to the acceptance of apkInspector. This timeline ensures that the decision to include apkInspector was made independently of our influence within Androguard, validating that the tool's acceptance was based on its own strengths and capabilities.

For the purposes of the tests described in the next section, we will be using a version of Androguard before the integration of apkInspector, to ensure that our findings are based on the tool's capabilities without the enhancements provided by our software.

5. Empirical Validation

5.1 Current Tooling

As we transition from the theoretical approach outlined in the chapter 4 to the empirical evidence of this chapter, it is essential to recognize the spark for this research. Initially, widely recognized tools within the Android security community, including those used in routine pentests and malware research, were impeded by sophisticated evasion techniques, leading to their failure in analyzing malicious APKs. This challenge was notably highlighted by Joe Security¹, whose early struggles with tampered APKs on their platform ignited the spark for this research.

In the following sections, we will present a detailed examination of how well-established tools such as Jadx, Apktool, ApkAnalyzer, and Androguard continue to struggle with these evasive techniques. Furthermore, we will introduce the performance of apkInspector, our newly developed tool, which was designed in response to these ongoing challenges. This comparison aims to highlight the advancements achieved by apkInspector against complex evasion methods.

5.1.1 jadx

As of April 2024, the most recent release of jadx is version 1.5.0, which we utilized for our testing. However, a review of the commit history indicates that there have been no updates specifically addressing evasion tactics. Therefore, it is likely that even older versions of jadx would have produced comparable results in our tests.

¹<https://x.com/joe4security/status/1674042511969468418>

```

L$ jadx-gui orig_infected.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings-on -Dswing.aatext=true
INFO - output directory: orig_infected
INFO - loading ...
ERROR - Failed to process zip file: /mnt/hgfs/share/Research/mal-samples/orig_infected.apk
jadx.core.utils.exceptions.JadxRuntimeException: Failed to process zip file: /mnt/hgfs/share/Research/mal-samples/orig_infected.apk
    at jadx.api.plugins.utils.ZipSecurity.visitZipEntries(ZipSecurity.java:148)
    at jadx.api.plugins.utils.ZipSecurity.readZipEntries(ZipSecurity.java:154)
    at jadx.plugins.input.dex.DexFileLoader.collectDexFromZip(DexFileLoader.java:89)
    at jadx.plugins.input.dex.DexFileLoader.load(DexFileLoader.java:72)
    at jadx.plugins.input.dex.DexFileLoader.loadDexFromFile(DexFileLoader.java:49)
    at java.base/java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:197)
    at java.base/java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:197)
    at java.base/java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1625)
    at java.base/java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:509)
    at java.base/java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:499)
    at java.base/java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:921)
    at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
    at jadx.plugins.input.dex.DexFileLoader.collectDexFiles(DexFileLoader.java:44)
    at jadx.plugins.input.dex.DexInputPlugin.loadFiles(DexInputPlugin.java:40)
    at jadx.plugins.input.dex.DexInputPlugin.loadFiles(DexInputPlugin.java:36)
    at jadx.api.JadxDecompiler.loadInputFiles(JadxDecompiler.java:153)
    at jadx.api.JadxDecompiler.load(JadxDecompiler.java:119)
    at jadx.gui.JadxWrapper.open(JadxWrapper.java:72)
    at jadx.gui.ui.MainWindow.lambda$loadFiles$0(MainWindow.java:520)
    at jadx.core.utils.tasks.TaskExecutor.wrapTask(TaskExecutor.java:166)
    at jadx.core.utils.tasks.TaskExecutor.runStages(TaskExecutor.java:142)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1136)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)
    at java.base/java.lang.Thread.run(Thread.java:833)
Caused by: java.util.zip.ZipException: invalid CEN header (bad compression method: 2920)
    at java.base/java.util.zip.ZipFile$Source.checkAndAddEntry(ZipFile.java:1728)
    at java.base/java.util.zip.ZipFile$Source.initCEN(ZipFile.java:1667)
    at java.base/java.util.zip.ZipFile$Source.<init>(ZipFile.java:1445)
    at java.base/java.util.zip.ZipFile$Source.get(ZipFile.java:1407)
    at java.base/java.util.zip.ZipFile$CleanableResource.<init>(ZipFile.java:716)
    at java.base/java.util.zip.ZipFile.<init>(ZipFile.java:250)
    at java.base/java.util.zip.ZipFile.<init>(ZipFile.java:179)
    at java.base/java.util.zip.ZipFile.<init>(ZipFile.java:193)
    at jadx.api.plugins.utils.ZipSecurity.visitZipEntries(ZipSecurity.java:130)
    ... 24 common frames omitted
ERROR - Failed to process zip file: orig_infected.apk
jadx.core.utils.exceptions.JadxRuntimeException: Failed to process zip file: /mnt/hgfs/share/Research/mal-samples/orig_infected.apk

```

Figure 5.1: jadx error when the original malware was processed.

Figure 5.1 illustrates jadx's failure to process the malware in its original form. The error, highlighted in the figure, displays the message `bad compression method`, indicating the core issue. This error arises because the compression method used in the malware is not typically supported by standard ZIP libraries, which are not designed to handle such edge cases.

```

L$ jadx-gui repacked_alignedINF.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings-on -Dswing.aatext=true
INFO - output directory: repacked_alignedINF
INFO - loading ...
INFO - Loaded classes: 76, methods: 701, instructions: 22090
ERROR - Failed to parse '.arsc' file
jadx.core.utils.exceptions.JadxException: Error decode: resources.arsc
    at jadx.api.ResourcesLoader.decodeStream(ResourcesLoader.java:79)
    at jadx.core.dex.nodes.RootNode.loadResources(RootNode.java:213)
    at jadx.api.JadxDecompiler.load(JadxDecompiler.java:127)
    at jadx.gui.JadxWrapper.open(JadxWrapper.java:72)
    at jadx.gui.ui.MainWindow.lambda$loadFiles$0(MainWindow.java:520)
    at jadx.core.utils.tasks.TaskExecutor.wrapTask(TaskExecutor.java:166)
    at jadx.core.utils.tasks.TaskExecutor.runStages(TaskExecutor.java:142)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1136)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)
    at java.base/java.lang.Thread.run(Thread.java:833)
Caused by: java.io.IOException: Decode error: Unexpected package header size, position: 0x120
    at jadx.core.xmlgen.CommonBinaryParser.die(CommonBinaryParser.java:39)
    at jadx.core.xmlgen.ResTableParser.parsePackage(ResTableParser.java:128)
    at jadx.core.xmlgen.ResTableParser.decodeTableChunk(ResTableParser.java:119)
    at jadx.core.xmlgen.ResDecoder.decode(ResDecoder.java:28)
    at jadx.core.dex.nodes.RootNode.lambda$loadResources$7(RootNode.java:213)
    at jadx.api.ResourcesLoader.decodeStream(ResourcesLoader.java:74)
    ... 9 common frames omitted
INFO - Resetting disk code cache, base dir: /home/kali/.cache/jadx/projects/repacked_alignedINF-2781cea75e2f5c56f0c45920694707bd/code
ERROR - Decode error
jadx.core.utils.exceptions.JadxException: Error decode: AndroidManifest.xml
    at jadx.api.ResourcesLoader.decodeStream(ResourcesLoader.java:79)

```

*repacked_alignedINF - jadx-gui

```

File View Navigation Tools Plugins Help
repacked_alignedINF. AndroidManifest.xml x
  Inputs
  Files
  Scripts
1 Error decode manifest
2 java.lang.IndexOutOfBoundsException
3 at java.base/java.nio.Buffer.checkIndex(Buffer.java:749)

```

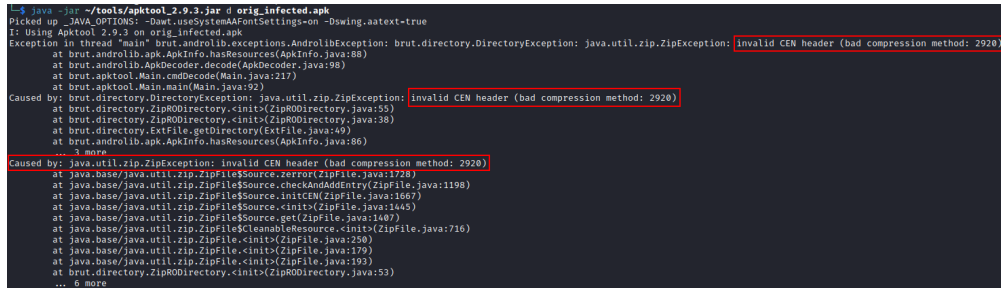
Figure 5.2: jadx error for both stage two and stage three tests

Figure 5.2 demonstrates that jadx fails to process the APK in both scenarios: when

the type identifier is fixed and when it is not (see Section 3.2.1). This suggests that jadx either does not validate the header type identifier, or it is sufficiently flexible to attempt processing the AXML despite a mismatching type identifier. Consequently, jadx progresses past the type identifier issue, but ultimately fails due to subsequent evasion tactics employed by the malware.

5.1.2 apktool

As of April 2024, the latest version of apktool available is 2.9.3. A review of the tool's commit history and recent issues indicates that it currently lacks the capability to handle these evasive tactics. This limitation has been documented as well in the reported issue².



```
~$ java -jar ~/tools/apktool_2.9.3.jar d orig_infected.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Is using Apktool 2.9.3 on orig_infected.apk
Exception in thread "main" brut.androlib.exceptions.AndrolibException: brut.directory.DirectoryException: java.util.zip.ZipException: invalid CEN header (bad compression method: 2920)
    at brut.androlib.apk.ApkInfo.hasResources(ApkInfo.java:88)
    at brut.androlib.ApkDecoder.decode(ApkDecoder.java:98)
    at brut.apktool.Main.cmdDecode(Main.java:217)
    at brut.apktool.Main.main(Main.java:92)
Caused by: brut.directory.DirectoryException: java.util.zip.ZipException: invalid CEN header (bad compression method: 2920)
    at brut.directory.ZipRODirectory.<init>(ZipRODirectory.java:55)
    at brut.directory.ZipRODirectory.<init>(ZipRODirectory.java:38)
    at brut.directory.ExtFile.getDirectory(ExtFile.java:49)
    at brut.androlib.apk.ApkInfo.hasResources(ApkInfo.java:86)
    ... 3 more
Caused by: java.util.zip.ZipException: invalid CEN header (bad compression method: 2920)
    at java.base/java.util.zip.ZipFile$Source.error(ZipFile.java:172)
    at java.base/java.util.zip.ZipFile$Source.checkAndAddEntry(ZipFile.java:1198)
    at java.base/java.util.zip.ZipFile$Source.initCEN(ZipFile.java:1667)
    at java.base/java.util.zip.ZipFile$Source.<init>(ZipFile.java:1445)
    at java.base/java.util.zip.ZipFile$Source.get(ZipFile.java:1407)
    at java.base/java.util.zip.ZipFile$CleanableResource.<init>(ZipFile.java:716)
    at java.base/java.util.zip.ZipFile.<init>(ZipFile.java:250)
    at java.base/java.util.zip.ZipFile.<init>(ZipFile.java:179)
    at java.base/java.util.zip.ZipFile.<init>(ZipFile.java:193)
    at brut.directory.ZipRODirectory.<init>(ZipRODirectory.java:53)
    ... 6 more
```

Figure 5.3: Apktool error when the original malware was processed.

Figure 5.3 illustrates apktool's failure to process the malware in its original form. Similar to jadx, the error displayed is associated with the compression method used, indicating that the tool is unable to proceed beyond this point. This issue is anticipated given that the ZIP library employed by apktool is not designed to manage such evasive tactics.

²<https://github.com/iBotPeaches/Apktool/issues/3540>

```

└─$ java -jar ~/tools/apktool_2.9.3.jar d repacked_alignedINF.apk
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
I: Using Apktool 2.9.3 on repacked_alignedINF.apk
I: Loading resource table ...
S: Unknown chunk type: 006c
I: Decoding file-resources ...
I: Decoding values */* XMLs ...
I: Decoding AndroidManifest.xml with resources ...
W: Bad string block: string entry is at 560, past end at 560
I: Loading resource table from file: /home/kali/.local/share/apktool/framework/1.apk
W: Unknown chunk type at: (0x00001590) skipping ...
W: Unknown chunk type at: (0x00001598) skipping ...
W: Unknown chunk type at: (0x000015a0) skipping ...
W: Unknown chunk type at: (0x0000455c) skipping ...
W: XML hit unexpected end of file at byte: 0x455c
I: Regular manifest package ...
[Fatal Error] :3:62: XML document structures must start and end within the same entity.
I: Baksmaling classes.dex ...
I: Copying assets and libs ...
I: Copying unknown files ...
I: Copying original files ...

(kali@kali)-[/mnt/hgfs/share/Research/mal-samples]
└─$ cat repacked_alignedINF/AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="wyija.utykuvr.uwpexgh" platformBuildVersionCode="23" platformBuildVersionName="6.0-2438415" a
ndroid:versionCode="1" android:versionName="17.1.22" android:compileSdkVersion="23" android:compileSdkVersionCod
ename="6.0-2438415" android:tag=""
xmlns:android="http://schemas.android.com/apk/res/android">

```

Figure 5.4: Apktool error for both stage two and stage three tests.

Figure 5.4 demonstrates that apktool fails to process the APK in both scenarios: when the type identifier is fixed and when it is not (see Section 3.2.1). This suggests that apktool either does not validate the header type identifier, or it is sufficiently flexible to attempt processing the AXML despite a mismatching type identifier. Consequently, apktool progresses past the type identifier issue, but ultimately fails due to subsequent evasion tactics employed by the malware.

5.1.3 apk-analyzer

The command line tools version `commandlinetools-linux-11076708_latest.zip` was used for these tests. For clarity the GUI import through the Android Studio was chosen instead of the CLI. Figure 5.5 shows the tool failing to open the malware in the original form, showing an identical error to the previous two tools.

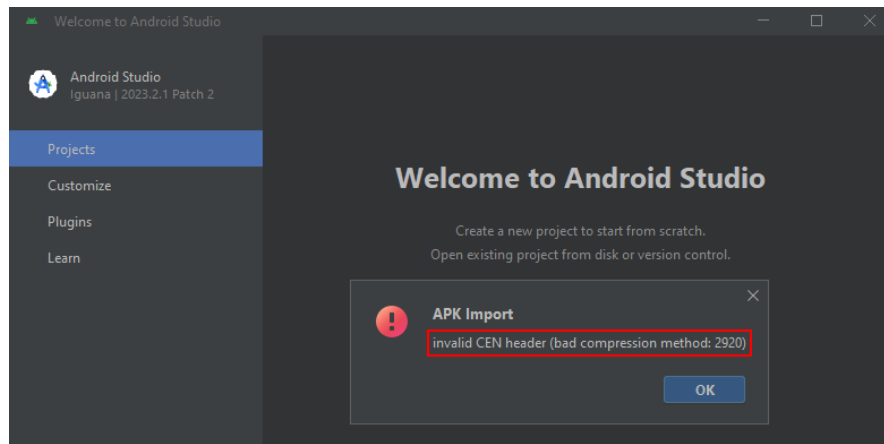


Figure 5.5: Apkanalyzer error while opening the original malware APK

For the stage two test, figure 5.6 shows an assertion error, which is high likely related to the fact that the header type identifier is not the expected one.

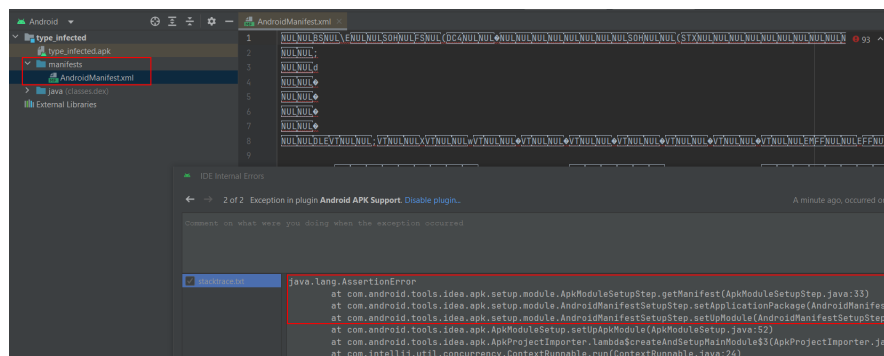


Figure 5.6: Apkanalyzer assertion error with a modified type identifier header.

In cases where the type identifier is corrected but other AXML evasion tactics remain, Figure 5.7 shows the APK loading, but the AndroidManifest.xml is not parsed correctly. The displayed error, an `IndexOutOfBoundsException`, is likely caused by dummy data in between the manifest elements (see Section 3.2.4). When the tool attempts to process this data assuming it is the header of the next element, it reads an excessive size as the element size, leading to the exception.

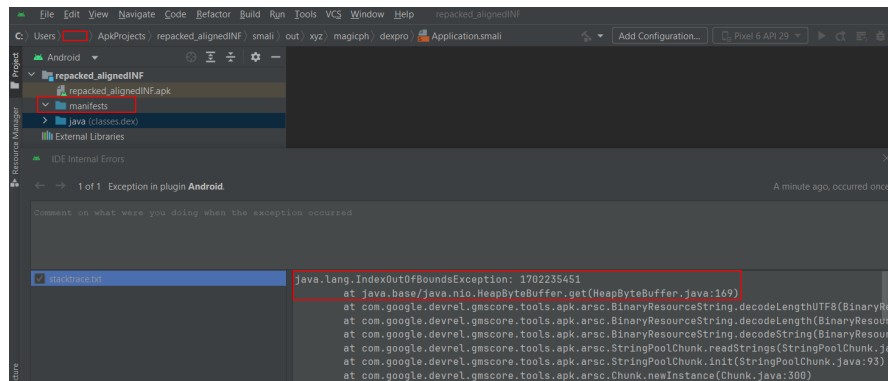


Figure 5.7: Apkanalyzer could open the APK file, but no manifest entry was presented.

5.1.4 Androguard

For the reasons explained in Section 4.4.2 version 3.4.0a1³ of androguard was used for the tests described below.

Figure 5.8 shows androguard failing to process the malware in the original form due to a decompression error, as expected.

```

--$ androguard decompile orig_infected.apk -o output
[INFO ] androguard.apk: Starting analysis on AndroidManifest.xml
Traceback (most recent call last):
  File "/home/kali/.local/bin/androguard", line 8, in <module>
    sys.exit(entry_point())
    ~~~~~^~~~~
  File "/usr/lib/python3/dist-packages/click/core.py", line 1157, in __call__
    return self.main(*args, **kwargs)
    ~~~~~^~~~~
  File "/usr/lib/python3/dist-packages/click/core.py", line 1078, in main
    rv = self.invoke(ctx)
    ~~~^~~~~
  File "/usr/lib/python3/dist-packages/click/core.py", line 1688, in invoke
    return _process_result(sub_ctx.command.invoke(sub_ctx))
    ~~~~~^~~~~
  File "/usr/lib/python3/dist-packages/click/core.py", line 1434, in invoke
    return ctx.invoke(self.callback, **ctx.params)
    ~~~~~^~~~~
  File "/usr/lib/python3/dist-packages/click/core.py", line 783, in invoke
    return __callback(*args, **kwargs)
    ~~~~~^~~~~
  File "/home/kali/.local/lib/python3.11/site-packages/androguard/cli/entry_points.py", line 395, in decompile
    s.add(fname, fd.read())
    ~~~~~^~~~~
  File "/home/kali/.local/lib/python3.11/site-packages/androguard/session.py", line 313, in add
    digest, _ = self.addAPK(filename, raw_data)
    ~~~~~^~~~~
  File "/home/kali/.local/lib/python3.11/site-packages/androguard/session.py", line 193, in addAPK
    apk = APK(data, True)
    ~~~~~^~~~~
  File "/home/kali/.local/lib/python3.11/site-packages/androguard/core/bytecodes/apk.py", line 291, in __init__
    self._apk_analysis()
  File "/home/kali/.local/lib/python3.11/site-packages/androguard/core/bytecodes/apk.py", line 311, in _apk_analysis
    manifest_data = self.zip.read(1)
    ~~~~~^~~~~
  File "/usr/lib/python3.11/zipfile.py", line 1518, in read
    with self.open(name, "r", pwd) as fp:
    ~~~~~^~~~~
  File "/usr/lib/python3.11/zipfile.py", line 1618, in open
    return ZipExtFile(zef_file, mode, zinfo, pwd, True)
    ~~~~~^~~~~
  File "/usr/lib/python3.11/zipfile.py", line 832, in __init__
    self._decompressor = _get_decompressor(self._compress_type)
    ~~~~~^~~~~
  File "/usr/lib/python3.11/zipfile.py", line 731, in _get_decompressor
    _check_compression(compress_type)
  File "/usr/lib/python3.11/zipfile.py", line 711, in _check_compression
    raise NotImplementedError("That compression method is not supported")
NotImplementedError: That compression method is not supported

```

Figure 5.8: Androguard error when the original malware was processed.

³<https://pypi.org/project/androguard/3.4.0a1/>

Androguard already included the capability to be tolerant against a tampered type identifier in the header of the AndroidManifest.xml file⁴. It even informed the user with the message "AXML file has an unusual resource type! Malware like to do such stuff....". Figure 5.9 shows this attempt.

```

L-$ androguard decompile type_infected.apk -o output
[INFO ] androguard.apk: Starting analysis on AndroidManifest.xml
[WARNING ] androguard.axml: AXML file has an unusual resource type! Malware likes to to such stuff to anti androguard! But we try to parse it anyways. Resource Type: 0x0000
[ERROR ] androguard.axml: Error parsing resource header: declared header size is smaller than required size of 8! Offset=5492
[ERROR ] androguard.apk: Error while parsing AndroidManifest.xml - is the file valid?
[INFO ] androguard.analysis: Adding DEX file version 35
[INFO ] androguard.analysis: Reading bytecode took : 0min 00s
[INFO ] androguard.analysis: End of creating cross references (XREF) run time: 0min 00s
Dump information type_infected.apk in output
Create directory output
Decompilation ... End
Dump Landroid/support/coreui/app/; <clinit> ()V ... source codes ... bytecodes ...
Dump Landroid/support/coreui/app/; <init> ()V ... bytecodes ...
Dump Landroid/support/coreui/app/E; <init> ()V ... source codes ... bytecodes ...
Dump Landroid/support/coreui/app/E; a (Ljava/lang/Class; Ljava/lang/Object; Ljava/lang/String;)Ljava/lang/Object; ... bytecodes
...

```

Figure 5.9: Androguard error when the sample with AXML evasion techniques was processed.

Figure 5.10 illustrates that Androguard encounters the same error as depicted in Figure 5.9. This observation aligns with expectations, as Androguard successfully bypasses the mismatch in the type identifier header but ultimately fails to parse the manifest completely. This failure likely occurs because the tool interprets dummy data among the elements as headers for subsequent elements.

```

L-$ androguard decompile repacked_alignedINF.apk -o output
[INFO ] androguard.apk: Starting analysis on AndroidManifest.xml
[ERROR ] androguard.axml: Error parsing resource header: declared header size is smaller than required size of 8! Offset=5492
[ERROR ] androguard.apk: Error while parsing AndroidManifest.xml - is the file valid?
[INFO ] androguard.analysis: Adding DEX file version 35
[INFO ] androguard.analysis: Reading bytecode took : 0min 00s
[INFO ] androguard.analysis: End of creating cross references (XREF) run time: 0min 00s
Dump information repacked_alignedINF.apk in output
Create directory output
Decompilation ... End
Dump Landroid/support/coreui/app/; <clinit> ()V ... source codes ... bytecodes ...
Dump Landroid/support/coreui/app/; <init> ()V ... bytecodes ...
Dump Landroid/support/coreui/app/E; <init> ()V ... source codes ... bytecodes ...
Dump Landroid/support/coreui/app/E; a (Ljava/lang/Class; Ljava/lang/Object; Ljava/lang/String;)Ljava/lang/Object; ... bytecodes
...
Dump Landroid/support/coreui/app/E; a (Ljava/lang/Class; Ljava/lang/Object; Ljava/lang/String; [Ljava/lang/Object; [Ljava/lang/Class;)Ljava/lang/Object; ... bytecodes ...
Dump Landroid/support/coreui/app/E; a (Ljava/lang/String;)Ljava/lang/Object; ... bytecodes ...
Dump Landroid/support/coreui/app/E; a (Ljava/lang/String; Ljava/lang/Object; Ljava/lang/String;)Ljava/lang/Object; ... bytecodes
S ...

```

Figure 5.10: Androguard error when the sample with stage three modifications was processed.

5.2 apkInspector

For the subsequent tests the latest version 1.2.3 of apkInspector as of April 2024 was used.

⁴https://github.com/androguard/androguard/blob/42c84d6a517f4ed08fd6b7a60c1a701e2ef66a47/androguard/core/axml/__init__.py#L430

5.2.1 Three-Stage Evasion Technique Testing

Figures 5.11, 5.12 and 5.13 show apkInspector in CLI mode, using flag `-m` in order to extract and convert to a readable format the `AndroidManifest.xml` for each of the three stages of the test.

The screenshot shows the terminal output of apkInspector. At the top, it displays the command 'apkInspector -apk orig_infected.apk -m', the version '1.2.3', and the copyright 'Copyright 2023 erev0s <projects@erev0s.com>'. Below this, it states 'AndroidManifest was saved as: decoded_AndroidManifest.xml'. The bottom part of the image shows a snippet of the extracted XML code in a dark-themed editor. The XML starts with a manifest tag, followed by package, platformBuildVersionCode, platformBuildVersionName, and android:versionCode attributes. It then contains two uses-sdk tags with various attributes like android:tag and Unknown_Attribute_Name. The XML is color-coded with red for tags and green for attributes and values.

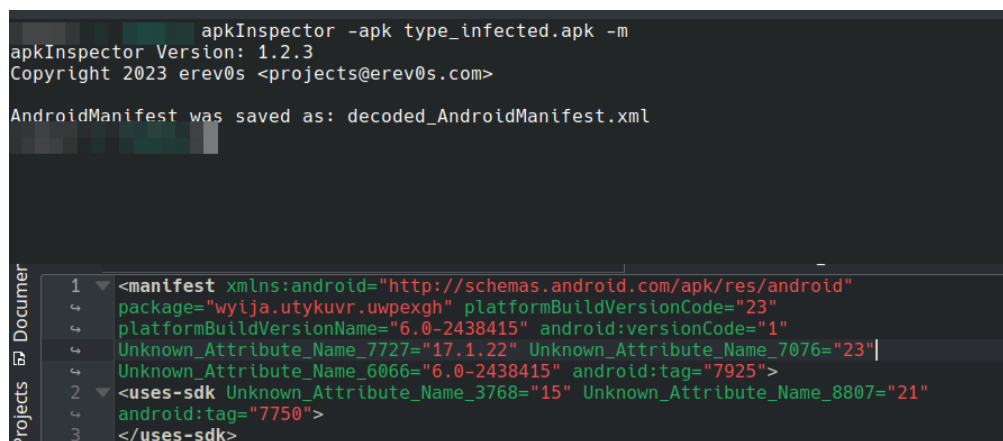
```
apkInspector -apk orig_infected.apk -m
apkInspector Version: 1.2.3
Copyright 2023 erev0s <projects@erev0s.com>

AndroidManifest was saved as: decoded_AndroidManifest.xml

1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="wyija.utykuvr.uwpexgh" platformBuildVersionCode="23"
  platformBuildVersionName="6.0-2438415" android:versionCode="1"
  Unknown_Attribute_Name_9167="17.1.22" Unknown_Attribute_Name_5639="23"
  Unknown_Attribute_Name_2572="6.0-2438415" android:tag="7925">
2 <uses-sdk Unknown_Attribute_Name_1911="15" Unknown_Attribute_Name_6854="21"
  android:tag="7750">
3 </uses-sdk>
```

Figure 5.11: apkInspector extracting and converting `AndroidManifest.xml` from the original state of the malware.

apkInspector effectively bypasses the evasion tactics associated with the ZIP structure by adopting an approach similar to that described in the Android source code (see Section 2.2.1). Consequently, alterations in compression methods or file sizes do not affect its functionality.

This screenshot is similar to the previous one, showing the terminal output of apkInspector for a different file, 'type_infected.apk'. It follows the same sequence: command execution, version/copyright display, and saving the manifest. The XML snippet at the bottom shows a different set of attributes, including Unknown_Attribute_Name_7727, Unknown_Attribute_Name_7076, and Unknown_Attribute_Name_6066, along with the same uses-sdk tag structure.

```
apkInspector -apk type_infected.apk -m
apkInspector Version: 1.2.3
Copyright 2023 erev0s <projects@erev0s.com>

AndroidManifest was saved as: decoded_AndroidManifest.xml

1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="wyija.utykuvr.uwpexgh" platformBuildVersionCode="23"
  platformBuildVersionName="6.0-2438415" android:versionCode="1"
  Unknown_Attribute_Name_7727="17.1.22" Unknown_Attribute_Name_7076="23"
  Unknown_Attribute_Name_6066="6.0-2438415" android:tag="7925">
2 <uses-sdk Unknown_Attribute_Name_3768="15" Unknown_Attribute_Name_8807="21"
  android:tag="7750">
3 </uses-sdk>
```

Figure 5.12: apkInspector extracting and converting `AndroidManifest.xml` with AXML evasion techniques.


```
apkInspector -apk repacked_alignedINF.apk -m
apkInspector Version: 1.2.3
Copyright 2023 erev0s <projects@erev0s.com>

AndroidManifest was saved as: decoded_AndroidManifest.xml

1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="wyija.utykuvr.uwpexgh" platformBuildVersionCode="23"
  platformBuildVersionName="6.0-2438415" android:versionCode="1"
  Unknown_Attribute_Name_4081="17.1.22" Unknown_Attribute_Name_3835="23"
  Unknown_Attribute_Name_8552="6.0-2438415" android:tag="7p25">
2 <uses-sdk Unknown_Attribute_Name_7723="15" Unknown_Attribute_Name_4183="21"
  android:tag="7750">
3 </uses-sdk>
```

Figure 5.13: apkInspector extracting and converting AndroidManifest.xml for stage three.

Figures 5.12 and 5.13 show apkInspector bypassing all the evasion tactics related to the AXML structure that this malware employs. As it can be seen, in all three cases, apkInspector was able to handle the extraction and conversion of the AXML to a human readable format without errors.

5.2.2 Play Store Application Testing for Accuracy

A dataset of 1809 applications was compiled and subjected to dual analysis using apkInspector and traditional tools like zipfile and androguard. The first phase involved extracting APK contents with both apkInspector and zipfile, comparing the resulting directories for consistency. The second phase focused on extracting and analyzing the AndroidManifest.xml files using apkInspector and androguard, with differences rigorously examined using xml.etree.ElementTree.

The results from testing apkInspector⁵ with a diverse set of applications from the Google Play Store demonstrate its high level of accuracy and reliability. Both apkInspector and traditional extraction methods successfully unzipped all 1809 applications, showcasing perfect alignment in basic functionality.

In the comparison of AndroidManifest.xml files, apkInspector and androguard yielded identical results in 1806 cases, based on the structure and entries of the XML. However, it is important to note that the content within each entry was not validated; apkIn-

⁵https://github.com/erev0s/apkInspector/tree/main/tests/top_apps

spector does not resolve attributes like `androguard` but rather translates the content it discovers. The minor discrepancies observed in three applications were due to different naming conventions for unknown system attributes, which do not affect the overall integrity of the manifests. These findings confirm that `apkInspector` performs on par with established tools in analyzing standard, non-malicious applications.

6. Conclusions

Throughout this study, we have thoroughly explored the persistent and evolving challenge posed by evasive techniques in Android malware, which are increasingly exploited in the digital landscape. Our detailed analysis has clarified the mechanics behind these techniques and pinpointed the deficiencies in current tooling that allow such methods to succeed undetected.

We identified a significant gap in the ability of existing tools to effectively handle and analyze Android malware employing these sophisticated evasion strategies. In response to this gap, we developed and introduced apkInspector, a tool specifically designed to address these challenges. apkInspector was tested against malware utilizing the same evasive techniques that hindered other tools, and it proved to be effective in overcoming obstacles that had previously resulted in errors. The integration of apkInspector into well-regarded tools like Androguard and Medusa further validates its effectiveness and acceptance within the security community, demonstrating its reliability and the trust it has garnered.

6.1 Future Work

Looking forward, there is a clear path for the continued development of apkInspector. As malware developers persistently innovate with new evasion techniques, apkInspector will evolve to counter these emerging threats. One potential area of expansion is the development of a Java version of apkInspector. Given that many applications and tools in the security and IT sectors are Java-based, a Java implementation would broaden the accessibility of apkInspector, allowing it to be seamlessly integrated as a library in numerous Java applications. This enhancement would not only increase apkInspector's utility but also its adoption, ensuring it remains at the forefront of technology in combating malware evasion techniques.

In addition to expanding its accessibility, we aim to enhance apkInspector with the capability to "fix" evasion methods in malware APKs. This would allow other tools,

which currently fail to parse such APKs, to successfully analyze them. Further, we plan to develop functionality within apkInspector that allows for the creation of evasive APKs. This feature will facilitate deeper research into evasion techniques, providing researchers and developers with the tools to test and strengthen their own systems against these tactics.

By committing to continuous improvement and adaptation, we aim to maintain apkInspector as a cutting-edge tool that meets the dynamic needs of the cybersecurity Android landscape, thereby safeguarding against modern evasive threats.

Bibliography

- [1] P. Inc. Appnote.txt - .zip file format specification, 2006. URL <https://pkwaredownloads.blob.core.windows.net/pkware-general/Documentation/APPNOTE-6.3.0.TXT>.
- [2] B. Krebs. How malicious android apps slip into disguise, 2023. URL <https://krebsonsecurity.com/2023/08/how-malicious-android-apps-slip-into-disguise/>.
- [3] G. R. Panakkal. Leaving our zip undone:how to abuse zip to deliver malware apps, 2014. URL <https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Panakkal.pdf>.
- [4] F. L. T. Research. Android/spynote moves to crypto currencies, 2024. URL <https://www.fortinet.com/blog/threat-research/android-spynote-moves-to-crypto-currencies>.
- [5] D. K. . Securelist. Soumnibot: the new android banker's unique techniques, 2024. URL <https://securelist.com/soumnibot-android-banker-obfuscates-app-manifest/112334/>.
- [6] H. Wang, H. Liu, X. Xiao, G. Meng, and Y. Guo. Characterizing android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 280–292, 2019. doi: 10.1109/ASE.2019.00035.
- [7] F. O. . Zimperium. Over 3,000 android malware samples using multiple techniques to bypass detection, 2023. URL <https://www.zimperium.com/blog/over-3000-android-malware-samples-using-multiple-techniques-to-bypass-detection/>.