SoumniBot: the new Android banker’s unique techniques

By Dmitry Kalinin

Published: 2024-04-17 · Archived: 2026-04-05 15:10:39 UTC

The creators of widespread malware programs often employ various tools that hinder code detection and analysis, and Android malware is no exception. As an example of this, droppers, such as Badpack and Hqwar, designed for stealthily delivering Trojan bankers or spyware to smartphones, are very popular among malicious actors who attack mobile devices. That said, we recently discovered a new banker, SoumniBot, which targets Korean users and is notable for an unconventional approach to evading analysis and detection, namely obfuscation of the Android manifest.

SoumniBot obfuscation: exploiting bugs in the Android manifest extraction and parsing procedure

Any APK file is a ZIP archive with AndroidManifest.xml in the root folder. This file contains information about the declared components, permissions and other app data, and helps the operating system to retrieve information about various app entry points. Just like the operating system, the analyst starts by inspecting the manifest to find the entry points, which is where code analysis should start. This is likely what motivated the developers of SoumniBot to research the implementation of the manifest parsing and extracion routine, where they found several interesting opportunities to obfuscate APKs.

Technique 1: Invalid Compression method value

This is a relatively well-known technique used by various types of malware including SoumniBot and associated with the way manifests are unpacked. In *libziparchive* library, the standard unarchiving function permits only two *Compression method* values in the record header: 0x0000 (STORED, that is uncompressed) и 0x0008 (DEFLATED, that is compressed with *deflate* from the *zlib* library), or else it returns an error.

```
static int32_t extractToWriter(ZipArchiveHandle handle, const ZipEntry64* entry,
                               zip_archive::Writer* writer) {
  const uint16_t method = entry->method;

  // this should default to kUnknownCompressionMethod.
  int32_t return_value = -1;
  uint64_t crc = 0;
  if (method == kCompressStored) {
    return_value =
        CopyEntryToWriter(handle->mapped_zip, entry, writer, kCrcChecksEnabled ? &crc : nullptr);
  } else if (method == kCompressDeflated) {
    return_value =
        InflateEntryToWriter(handle->mapped_zip, entry, writer, kCrcChecksEnabled ? &crc : nullptr);
  }

  if (!return_value && entry->has_data_descriptor) {
    return_value = ValidateDataDescriptor(handle->mapped_zip, entry);
    if (return_value) {
      return return_value;
    }
  }

  // Validate that the CRC matches the calculated value.
  if (kCrcChecksEnabled && (entry->crc32 != static_cast<uint32_t>(crc))) {
    ALOGW("Zip: crc mismatch: expected %" PRIu32 ", was %" PRIu64, entry->crc32, crc);
    return kInconsistentInformation;
  }

  return return_value;
}
```

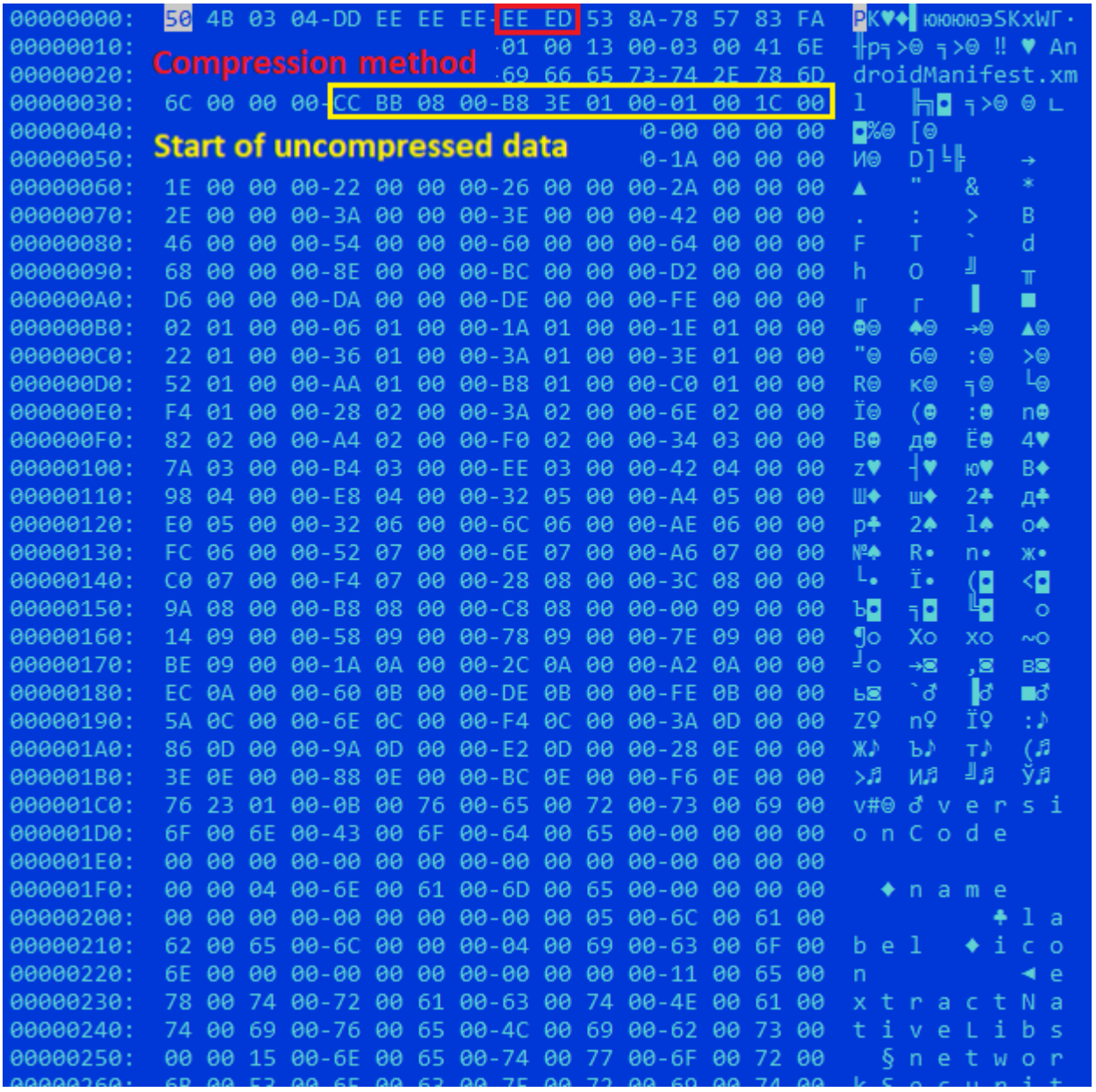
libziparchive unarchiving algorithm

Yet, instead of using this function, the developers of Android chose to implement an alternate scenario, where the value of the *Compression method* field is validated incorrectly.

```
if (entry.method == kCompressDeflated) {
  if (!asset_map.Create(fd, entry.offset + fd_offset, entry.compressed_length,
                        name_.GetDebugName().c_str(), incremental_hardening)) {
    LOG(ERROR) << "Failed to mmap file '" << path << "' in APK '" << name_.GetDebugName()
               << "'";
    return {};
  }

  std::unique_ptr<Asset> asset =
      Asset::createFromCompressedMap(std::move(asset_map), entry.uncompressed_length, mode);
  if (asset == nullptr) {
    LOG(ERROR) << "Failed to decompress '" << path << "' in APK '" << name_.GetDebugName()
               << "'";
    return {};
  }
  return asset;
}

if (!asset_map.Create(fd, entry.offset + fd_offset, entry.uncompressed_length,
                      name_.GetDebugName().c_str(), incremental_hardening)) {
  LOG(ERROR) << "Failed to mmap file '" << path << "' in APK '" << name_.GetDebugName() << "'";
  return {};
}

base::unique_fd ufd;
if (name_.GetPath() == nullptr) {
  // If the zip name does not represent a path, create a new `fd` for the new Asset to own in
  // order to create new file descriptors using Asset::openFileDescriptor. If the zip name is a
  // path, it will be used to create new file descriptors.
  ufd = base::unique_fd(dup(fd));
  if (!ufd.ok()) {
    LOG(ERROR) << "Unable to dup fd '" << path << "' in APK '" << name_.GetDebugName() << "'";
    return {};
  }
}

auto asset = Asset::createFromUncompressedMap(std::move(asset_map), mode, std::move(ufd));
if (asset == nullptr) {
  LOG(ERROR) << "Failed to mmap file '" << path << "' in APK '" << name_.GetDebugName() << "'";
  return {};
}
return asset;
```

Manifest extraction procedure

If the APK parser comes across any *Compression method* value but 0x0008 (DEFLATED) in the APK for the AndroidManifest.xml entry, it considers the data uncompressed. This allows app developers to put any value except 8 into *Compression method* and write uncompressed data. Although any unpacker that correctly implements compression method validation would consider a manifest like that invalid, the Android APK parser recognizes it correctly and allows the application to be installed. The image below illustrates the way the technique is executed in the file [b456430b4ed0879271e6164a7c0e4f6e](b456430b4ed0879271e6164a7c0e4f6e).



Invalid Compression method value followed by uncompressed data

Technique 2: Invalid manifest size

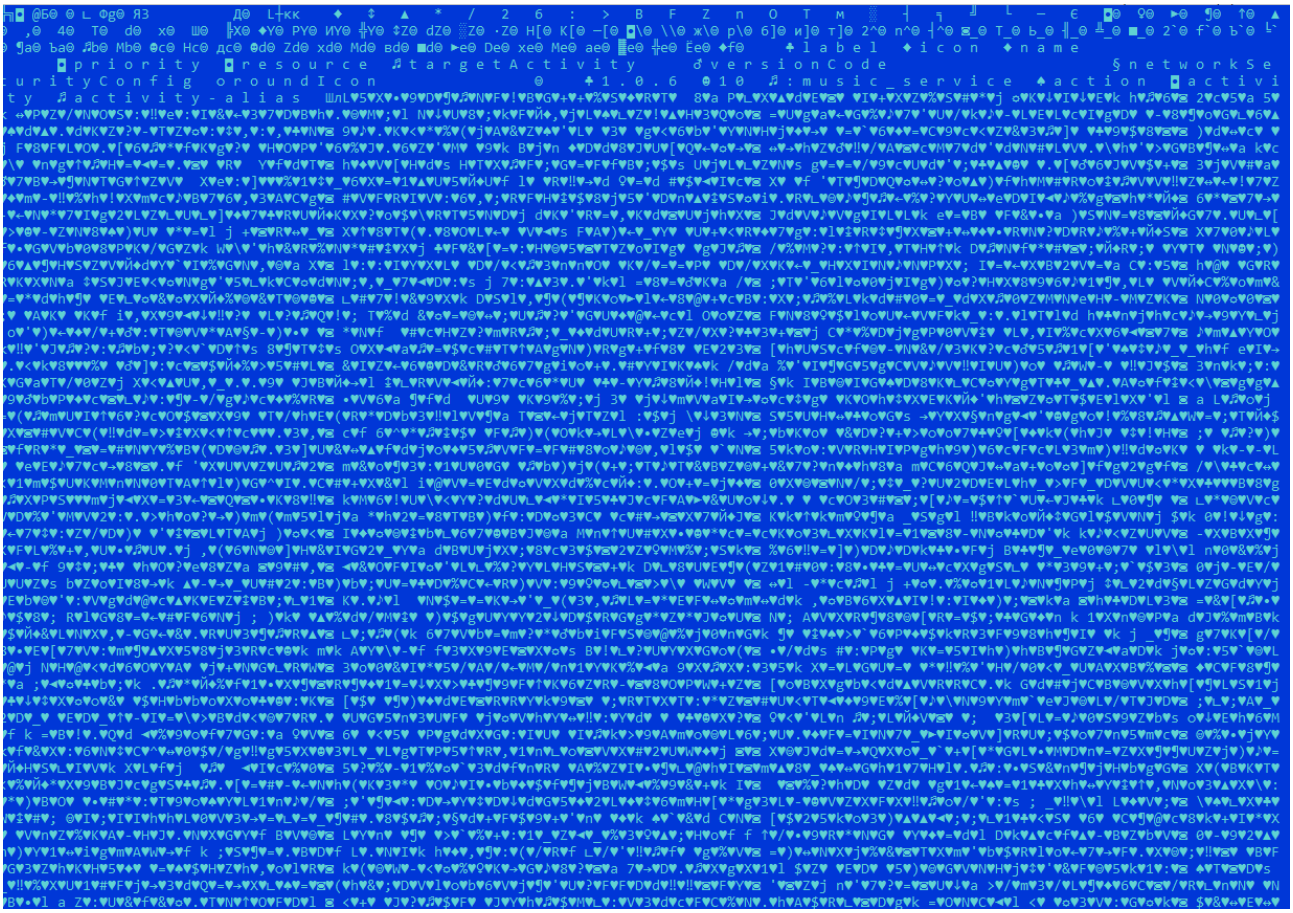
Let's use the file [0318b7b906e9a34427bf6bbcf64b6fc8] as an example to review the essence of this technique. The header of AndroidManifest.xml entry inside the ZIP archive states the size of the manifest file. If the entry is stored uncompressed, it will be copied from the archive unchanged, even if its size is stated incorrectly. The manifest parser ignores any overlay, that is information following the payload that's unrelated to the manifest. The malware takes advantage of this: the size of the archived manifest stated in it exceeds its actual size, which results in overlay, with some of the archive content being added to the unpacked manifest. Stricter manifest parsers wouldn't be able to read a file like that, whereas the Android parser handles the invalid manifest without any errors.

The stated size of the manifest is much larger than its actual size

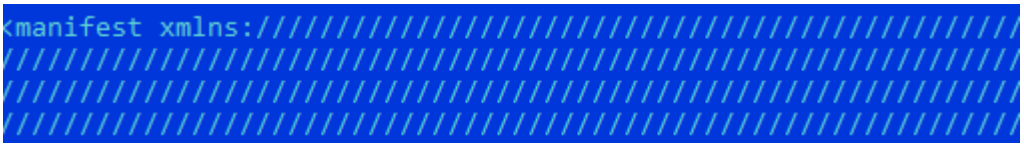
Note that although live devices interpret these files as valid, apkanalyzer, Google's own official utility for analyzing assembled APKs, cannot handle them. We have notified Google accordingly.

Technique 3: Long namespace names

The SoumniBot malware family, for example the file fa8b1592c9cda268d8affb6bceb7a120, has used this technique as well. The manifest contains very long strings, used as the names of XML namespaces.



Very long strings in the manifest…



…used as namespace names

Manifests that contain strings like these become unreadable for both humans and programs, with the latter may not be able to allocate enough memory to process them. The manifest parser in the OS itself completely ignores namespaces, so the manifest is handled without errors.

What’s under the obfuscation: SoumniBot’s functionality

When started, the application requests a configuration with two parameters, mainsite и mqtt, from the server, whose address being a hardcoded constant.

```
public void b(n.d d0, n.t t0) {
    try {
        s.log(("execute.isSuccessful()>>" + t0.d()));
        if(t0.d()) {
            JSONObject jSONObject0 = new JSONObject(((h0)t0.a()).x0());
            String s = jSONObject0.getString("mainsite");
            String s1 = jSONObject0.getString("mqtt");
            wgkx.wuar.jbkl.c.g.mainsite = s;
            k.mqttServer = s1.split(":")[0];
            k.mqttPort = Integer.parseInt(s1.split(":")[1]);
            return;
        }

        return;
    }
    catch(Exception exception0) {
        throw new RuntimeException(exception0);
    }
}
```

Parameter request

Both parameters are server addresses, which the malware needs for proper functioning. The mainsite server receives collected data, and mqtt provides MQTT messaging functionality for receiving commands. If the source server did not provide these parameters for some reason, the application will use the default addresses, also stored in the code.

After requesting the parameters, the application starts a malicious service. If it cannot start or stops for some reason, a new attempt is made every 16 minutes. When run for the first time, the Trojan hides the app icon to complicate removal, and then starts to upload data in the background from the victim's device to mainsite every 15 seconds. The data includes the IP address, country deduced from that, contact and account lists, SMS and MMS messages, and the victim's ID generated with the help of the trustdevice-android library. The Trojan also subscribes to messages from the MQTT server to receive the commands described below.

#	**Description**	**Parameters**
0	Sends information about the infected device: phone number, carrier, etc., and the Trojan version, followed by all of the victim's SMS messages, contacts, accounts, photos, videos and online banking digital certificates.	–
1	Sends the victim's contact list.	–
2	Deletes a contact on the victim's device.	*data*: the name of the contact to delete
3	Sends the victim's SMS and MMS messages.	–
4	A debugging command likely to be replaced with sending call logs in a new version.	–
5	Sends the victim's photos and videos.	–

8	Sends an SMS message.	*data*: ID that the malware uses to receive a message to forward. The Trojan sends the ID to mainsite and gets message text in return.
24	Sends a list of installed apps.	–
30	Adds a new contact on the device.	*name*: contact name; *phoneNum*: phone number
41	Gets ringtone volume levels.	–
42	Turns silent mode on or off.	*data*: a flag set to 1 to turn on silent mode and to 0 to turn it off
99	Sends a *pong* message in response to an MQTT ping request.	–
100	Turns on debug mode.	–
101	Turns off debug mode.	–

The command with the number 0 is worth special mention. It searches, among other things, external storage media for .key and .der files that contain paths to /NPKI/yessign.

```
1  public static List getAllBankingKeys(Context context) {
2  List list = new ArrayList();
3  Cursor cursor = context.getContentResolver().query(MediaStore.Files.getContentUri("external"),
4  new String[]{"_id", "mime_type", "_size", "date_modified", "_data"},
5  "(_data LIKE \'%.key\' OR _data LIKE \'%.der\')", null, null);
6  int index = cursor == null ? 0 : cursor.getColumnIndexOrThrow("_data");
7  if (cursor != null) {
8  while (cursor.moveToNext()) {
9  String s = cursor.getString(index);
10 If (!s.contains("/NPKI/yessign")) {
11 continue;
12 }
```

```
13  Logger.log("path is:" + s);
14  list.add(s);
15  break;
16  }
17  cursor.close();
18  }
19  return list;
20  }
```

If the application finds files like that, it copies the directory where they are located into a ZIP archive and sends it to the C&C server. These files are digital certificates issued by Korean banks to their clients and used for signing in to online banking services or confirming banking transactions. This technique is quite uncommon for Android banking malware. Kaspersky security solutions detect SoumniBot despite its sophisticated obfuscation techniques, and assign to it the verdict of Trojan-Banker.AndroidOS.SoumniBot.

Conclusion

Malware creators seek to maximize the number of devices they infect without being noticed. This motivates them to look for new ways of complicating detection. The developers of SoumniBot unfortunately succeeded due to insufficiently strict validations in the Android manifest parser code.

We have detailed the techniques used by this Trojan, so that researchers around the world are aware of the tactics, which other types of malware might borrow in the future. Besides the unconventional obfuscation, SoumniBot is notable for stealing Korean online banking keys, which we rarely observe in Android bankers. This feature lets malicious actors empty unwitting victims' wallets and circumvent authentication methods used by banks. To avoid becoming a victim of malware like that, we recommend using a reliable security solution on your smartphone to detect the Trojan and prevent it from being installed despite all its tricks.

Indicators of compromise

MD5

0318b7b906e9a34427bf6bbcf64b6fc8
00aa9900205771b8c9e7927153b77cf2
b456430b4ed0879271e6164a7c0e4f6e
fa8b1592c9cda268d8affb6bceb7a120

C&C

https[://]google.kt9[.]site
https[://]dbdb.addea.workers[.]dev

Source: https://securelist.com/soumnibot-android-banker-obfuscates-app-manifest/112334/