Analyzing Android malware using a FortiSandbox

By Axelle Apvrille

Published: 2017-08-17 · Archived: 2026-04-02 12:46:11 UTC

In this blog post we will analyze a couple of **Android malware samples in the Android VM of the FortiSandbox**. We'll also share a few interesting and useful **tricks**.

Running a sample in the VM

To run a given sample in the Android VM, you should log into the FortiSandbox, make sure an Android VM is available, and then "Scan Input" / Submit a New File.

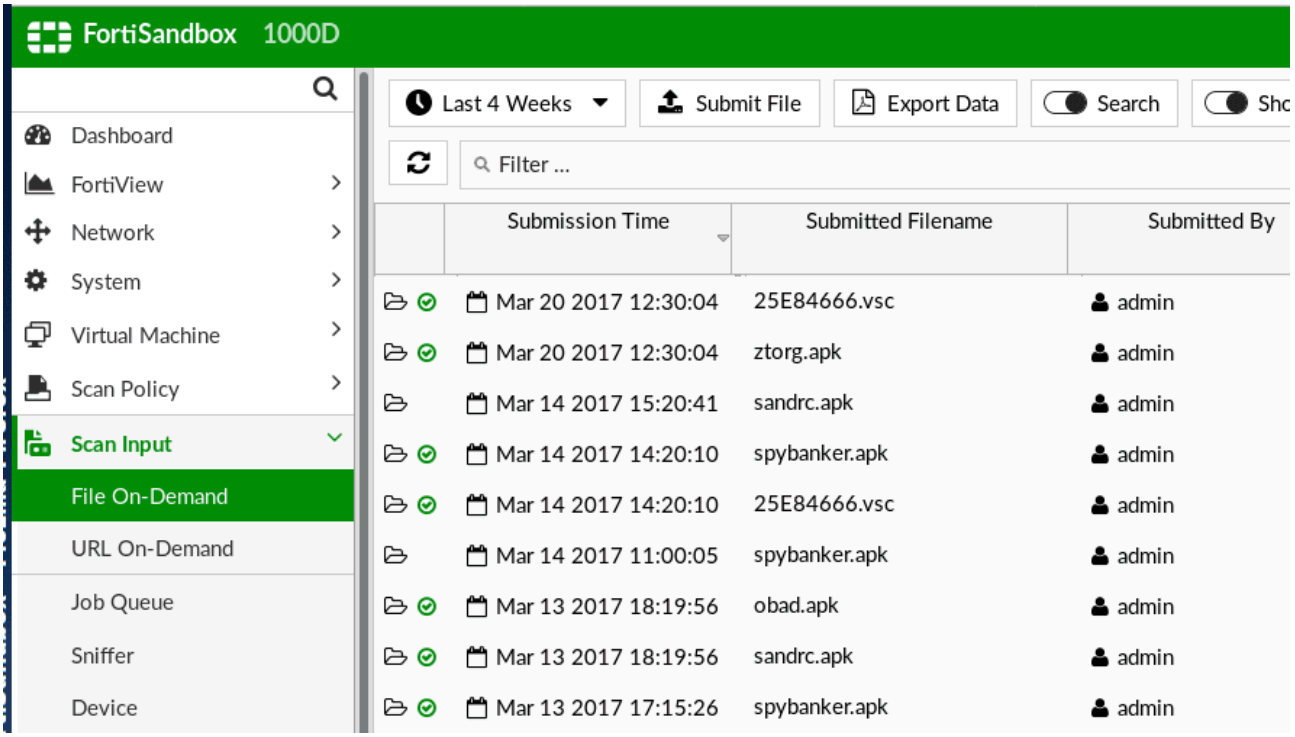

Figure 1: File On Demand

Next, if the objective is to *run the malware in the sandbox*, you must make sure to skip "static scan," "AV scan," and "Cloud Query" or they are likely to detect your malicious sample even before it reaches the sandbox.

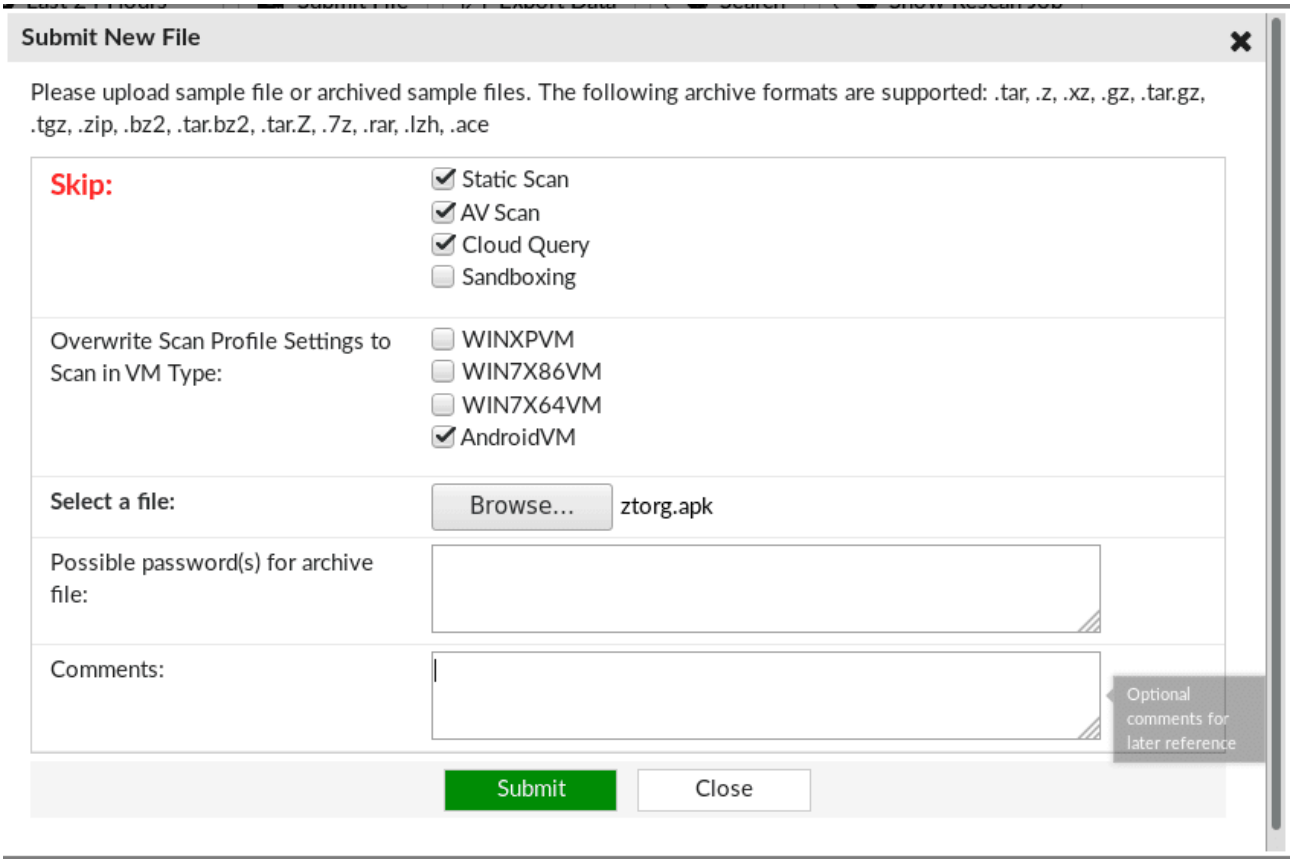

Figure 2: Skipping AV Scan

Samples analyzed:

Name	SHA256
Android/SpyBanker.DZ!tr	6d4ece4c5712995af7b76a03b535a3eaf10fcdca20f892f8dc9bdaf3fa85d590
Android/Obad.A!tr	ba1d6f317214d318b2a4e9a9663bc7ec867a6c845affecad1290fd717cc74f29
Android/Sandr.C!tr	29794b943cd398186be9f2ea59efc0ac698dcc213eea55cc64255913489e8d5c

Look in tracer.log

The sandbox outputs a tracer package which contains valuable information for analysis. In particular, the `tracer.log` file keeps track of process creation, events, and function calls and what they return. It is lengthy to read, but very precise.

```
I/FTNT    ( 1138): [1138]Call: void com.googie.system.MainActivity.onCreate(android.os.Bundle) -> public void a
```

- **I/FTNT**: tag for the Fortinet tracer.
- **1138**: process PID
- **Call / Return**: Call means we are calling a given method. Return means it is returning.

- `A -> B = C` : this means that method `A` calls method `B` . The precise call to `B` , with its argument values, is shown in statement `C` . If this is a return, `C` shows what is returned.

For example, the Android/SpyBanker malware opens a socket with `hxxp://193.201.224.22:3000`

```
 I/FTNT    ( 1138): [1138]Return: public void com.googie.system.SocketService.init() -> public java.lang.Object
...
I/FTNT    ( 1138): [1138]Call: public static io.socket.client.Socket io.socket.client.IO.socket(java.lang.String
```

Later, you will see a connection error on this socket (because the remote C&C no longer responds, of course):

```
 I/FTNT    ( 1138): [1138]Call: public io.socket.emitter.Emitter io.socket.emitter.Emitter.on(java.lang.String,
```

Handling SMS

As you may know, Android/SpyBanker spies on incoming SMS messages. Fortunately, this malicious behaviour is shown by the sandbox, which sends a few test SMS messages to the Android VM.

For example, the traces below show the malware processing an incoming SMS. We see the malware's function `getMessage()` gets called. It retrieves the SMS from the incoming PDU (first line), reads the originating phone number (second line), which is "+12345678" (third line). It then retrieves the message body (fourth line), which is "ping" (fifth line).

```
 Return: private com.googie.system.MessageItem com.googie.system.Receiver.getMessage(android.os.Bundle) -> publ
I/FTNT    ( 1138): [1367]Call: private com.googie.system.MessageItem com.googie.system.Receiver.getMessage(andro
I/FTNT    ( 1138): [1367]Return: private com.googie.system.MessageItem com.googie.system.Receiver.getMessage(and
I/FTNT    ( 1138): [1367]Call: private com.googie.system.MessageItem com.googie.system.Receiver.getMessage(andro
I/FTNT    ( 1138): [1367]Return: private com.googie.system.MessageItem com.googie.system.Receiver.getMessage(and
```

Listing malicious file activity in the sandbox

This feature is very useful because it makes it possible to list all the files the malware uses (creates, reads, or writes). The trick is to search the trace logs for any call to `sys_open` and then read the file name.

This bash snippet does wonders:

```
$ grep --only-matching -E "sys_open\(\".*\"," tracer.log | sed -e 's/sys_open("//g' | sed -e 's/",//g' | sort
```

This outputs several files, many of which correspond to the Android VM. For example, these are the relevant files for Android/Sandr.C:

- /data/app/net.droidjack.server-1.apk

- /data/dalvik-cache/data@app@net.droidjack.server-1.apk@classes.dex
- /data/data/net.droidjack.server/databases
- /data/data/net.droidjack.server/databases/SandroRat_Configuration_Database
- /data/data/net.droidjack.server/databases/SandroRat_Configuration_Database-journal
- /data/data/net.droidjack.server/databases/SandroRat_CrashReport_Database
- /data/data/net.droidjack.server/databases/SandroRat_CrashReport_Database-journal

Decrypting obfuscated strings

We can list all strings used by a malware with an adequate `grep` in the traces.

```
$ grep --only-matching -E "java.lang.String \".*\"" tracer.log
```

Good news! This works for any string the malware constructs, i.e also for **decrypted strings**.

For instance, Android/Obad.A!tr implements string obfuscation. In string obfuscated classes, there is an obfuscated static string table at the beginning of the class, and later a home-made decryption function named `cOIcOOo`.

The decryption function decrypts part of the string table. It takes three integers as parameters. One of these parameters resolves to the offset in the string table to start decrypting, and another resolves to the length to decrypt.

The inner implementation of the decryption function is slightly different for each class, so that a single decryption function cannot decrypt all strings.

One way to decrypt the strings is to *write a decryptor for each string obfuscated class, or a disassembler plugin. This works but takes some time to implement.*

A quicker solution consists in using the traces of the sandbox and reading the outputs for return calls to `cOIcOOo`. For example, in the sample below one string decrypts to "AES/CBC/PKCS5Padding":

```
I/FTNT    ( 1085): [1085]Return: static void com.android.system.admin.CIOIIolc.() -> private static java.lang.S
```

For a nicer output, we can `grep` through the traces to decrypt all strings that way:

```
$ grep -E "Return: .*cOIcOOo\(int,int,int\)\ =\ " tracer.log | sed -e 's/.*java.lang.String "//g' | sed -e 's/'
```

We get numerous **decrypted strings** such as:

```
 AES/CBC/PKCS5Padding
Blowfish/CBC/PKCS5Padding
android.os.Build
BOARD
```

```
BRAND
DEVICE
ID
MODEL
PRODUCT
getLine1Number
getSubscriberId
```

Defeating Reflection

Traces are also useful to work around reflection obfuscation tricks. For example, the following calls (from Android/Obad) the `connect()` method of `java.net.HttpURLConnection`.

```
 /FTNT    ( 1085): [1108]Call: private static byte[] com.android.system.admin.oIlclcIc.IoOoOIOI(java.lang.String
...
I/FTNT    ( 1085): [1108]Call: private static byte[] com.android.system.admin.oIlclcIc.IoOoOIOI(java.lang.String
```

Hope you enjoyed the tricks!

Thanks to Alain Forcioli who helped for this research.
