Playing Hide and Seek with Dalvik Executables

Axelle Apvrille, Fortinet, FortiGuard Labs
120, rue Albert Caquot 06410 Biot, France
aapvrille@fortinet.com*

Abstract

Android's Dalvik Executables (DEX) are full of sneaky corners, and this is just perfect for a game of Hide and Seek.

The first round of the game begins by hiding an entire method within a DEX file. The method hides so well that all common disassemblers (baksmali, apktool, Androguard, IDA Pro...) are unable to see it. Nevertheless, we show the method is still there as we can call it and execute it! The mechanism exploits a lack of verification of methods' layout and it is particulary convenient to hide a behaviour. Possible implications are the bypassing of market places' screening or anti-reversing of malware.

Then, like in the Hide and Seek game, the second round focuses on finding the hidden parts. The paper explains where to look for hidden data, and we provide a script to un-cloak the DEX file. The method shows back again.

The paper also discusses the PoC code and script that demo hiding and unhiding.

1 (Short) Introduction

This paper presents a trick which is capable of hiding parts of code in Android executables. The hidden parts are invisible (or difficult to see) to reverse engineers. A Proof of Concept application is detailed at section 7, as well as a helper tool in section 8.

The other sections focus on explaining how the tweak works. We provide some short background information on Android executables in section 2, keeping the focus on the only fields we need for the tweak. Then, we explain the concept of hiding methods in section 3. Sections 4 and 5 explain how to (re)package the modificated application: creating a valid patched executable and including it in an application. Hiding is only one part of the trick: we are also capable of actually invoking (i.e calling and using) the hidden method. We show how in section 6.

Finally, at the end, the paper discusses possible solutions to spot and unhide hidden parts (section 9).

2 Dalvik Executables

DEX files, *Dalvik Executables*, are the heart of Android applications: they contain the compiled classes of the applications, using Dalvik bytecode [AOSa] which is similar (but not exactly identical) to Java bytecode. Their format is defined in [AOSb] and our hide-and-seek technique consists in tweaking it.

We provide below a short overview of the structure of DEX files and important fields for hiding data. The most important parts of a DEX file are illustrated at Figure 1:

*see additional info for submission in section 11.

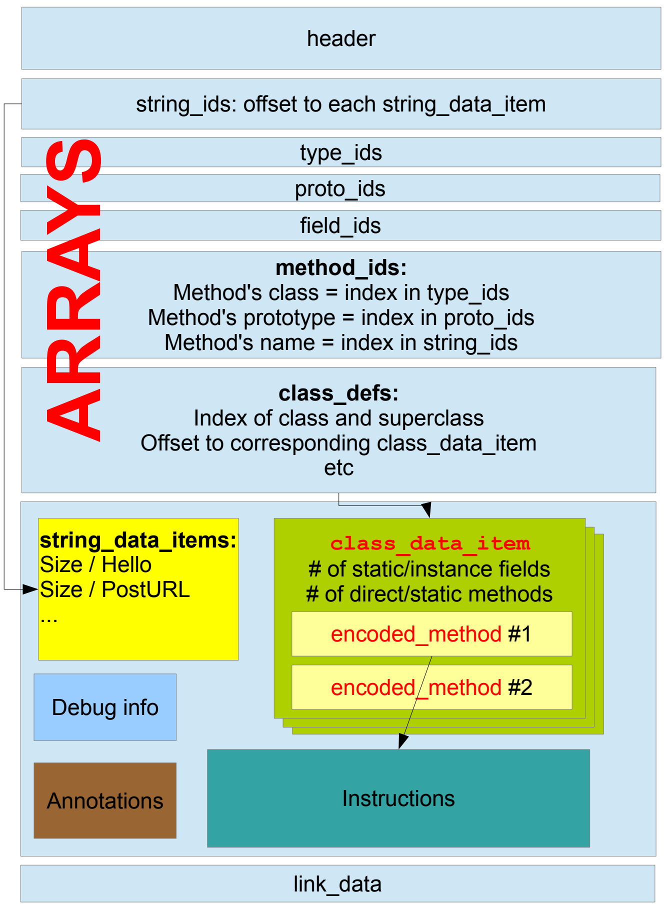

header
string_ids: offset to each string_data_item
type_ids
proto_ids
field_ids
method_ids:
Method's class = index in type_ids
Method's prototype = index in proto_ids
Method's name = index in string_ids
class_defs:
Index of class and superclass
Offset to corresponding class_data_item
etc
ARRAYS
string_data_items:
Size / Hello
Size / PostURL
...
class_data_item
# of static/instance fields
# of direct/static methods
encoded_method #1
encoded_method #2
Debug info
Annotations
Instructions
link_data

Figure 1: Main parts of a Dalvik Executable

- a Header - with file hash, checksum and sizes and offsets to parse the file
- a Data section, with the code instructions, the strings, fields, debug info etc. It can be seen as the core of the DEX file. The other sections of the file don't store any data, only offsets to data.
- Several arrays: an array of strings identifiers, of type identifiers, prototype identifiers, field identifiers, method identifiers and class identifiers. The entries of those arrays reference offsets to relevant data in the data section, or indexes to other arrays. For example, if a class is named "MrHyde", there is an entry in the `class_defs` array for that class. In that `class_defs` entry, the name of the class consists in an index in the strings array. The corresponding entry in the `string_ids` arary provides an offset to a `string_data_item` in the data section, which consists in the size of "MrHyde" followed by the string itself ("MrHyde").

Each class is described in the data section, by a structure named `class_data_item` which lists particular fields and methods of the class. The methods are organized in two different arrays: *direct methods* (static, private or constructor methods) and *virtual methods*(the other ones). Each method is declared in `class_data_item` as a structure named `encoded_method`, as follows:

- a **method_idx_diff**. [AOSb] defines this as "*index into the method_ids list for the identity of this method (includes the name and descriptor), represented as a difference from the index of previous element in the list. The index of the first element in a list is represented directly*". So, basically, this is an increment in indexes to entries of `method_ids`. If the previous method was reading index 3, and the next method has `method_idx_diff` = 1, then the index for this next method is $3 + 1 = 4$.
- an **access flag**. Typically `ACC_PUBLIC = 0x1` for public access. There are other flags such as `ACC_PRIVATE, ACC_PROTECTED, ACC_STATIC, ACC_FINAL` etc.
- **offset to the code**. The offset starts from the beginning of the DEX file.

3 Hiding methods

This section explains how to hide a given method of a given class in a Dalvik Executable. Globally, the steps are as follows:

1. **Patch the method's declaration in the DEX filein** its `encoded_method` entry. Basically, the patch consists in modifying the declaration of the method to hide, so that it points to *another existing method*. We detail the patch in this section.
2. **Re-computing the DEX file**: its SHA1 hash and Adler checksum must be re-computed and the DEX header fixed accordingly. We explain how to do this in Section 4.
3. **Re-build the APK** from the updated DEX file. This is detailed in Section 5.

Let's investigate how to patch the method's declaration. Actually, there are several alternatives. For instance, we can patch so that we *refer twice to the previous method*. This case is illustrated at Figure 2. It consists in the following modifications:

- **Set hidden method's idx to 0**. Consequently, the Dalvik verifier sees this method under the name and signature descriptor of the *previous* method.
- **No need to modify the access flag**. Actually, it is possible to change it if we want to (but usually there is no need to). We must ensure that the new flag does not change the method's category (direct, virtual). Precisely, this means that if we are patching a virtual method, the patched access flag must not include private, static or constructor. Reciprocally, if we are patching a direct method, be sure to have the access flag include one of those values.
  A wrong access flag prevents the application from being installed

  ```
  Failure [INSTALL_FAILED_DEXOPT]
  ```
- **Set code offset to offset of the previous method**. Using a bad code offset results in complaints from the verifier.

For the chaining to be complete, the declaration of the next method must also be patched: its idx must be increased to jump over the hidden method:

$$next\_method\_idx = original\_next\_method\_idx + original\_method\_idx \quad (1)$$

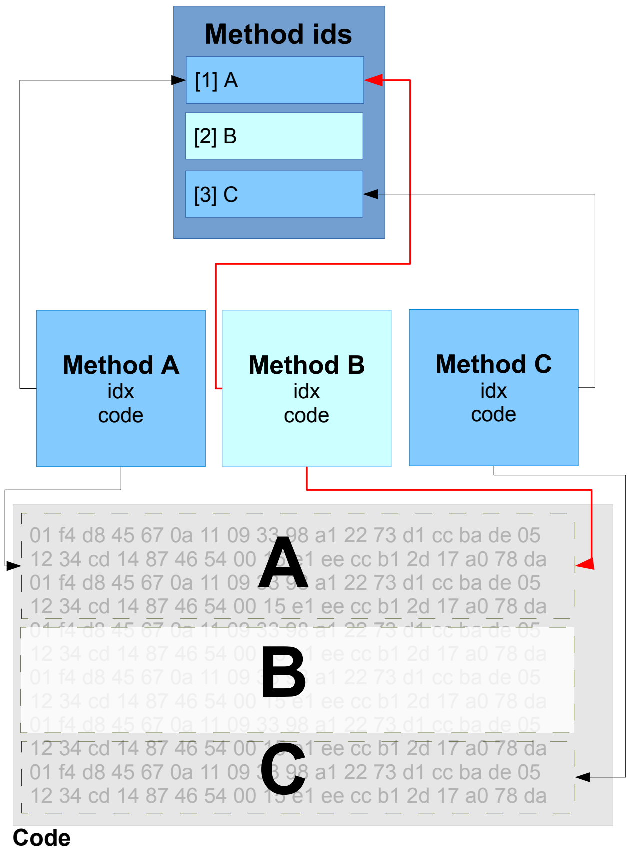


Figure 2: Hiding a method in a Dalvik Executable. The red links correspond to the patch to hide method B. Method A is referenced twice both for indexes and code. Method B is never referenced.

Description	Method_idx	Access flag	Code offset (2 bytes)	Next method_idx
Hide a method - reference twice previous method	0	keep	code offset for previous method	original_next_method_idx + original_method_idx
Hide a method - reference twice next method	original_next_method_idx + original_method_idx	keep	code offset for next method	0

Table 1: Two different techniques to hide a particular method in a Dalvik Executable

Instead of referring twice the previous method, it is also possible to r*efer twice the next method*. In that case, the hidden method's idx is incremented to jump over the hidden method, the code offset refers to the next method, and the idx of the next method is set to 0.

Note that in either cases the number of methods of class (direct_methods_size or virtual_methods_size) must not be changed. The class still carries the information for the hidden method. It's just that it does not show.

In an Android challenge of Hashdays 2012 [Apv12b], I demonstrated a very special case of method hiding: hiding the only virtual method of a class. In that particular situation, the hiding is simpler because we can purely erase the last entry: set the number of virtual methods of the class (`virtual_methods_size`) to 0, and nullify the method's entry: 0 for idx, 0 for access flag, 0 for code offset. Hiding any other method is however a bit more complicated as we have seen.

4 Building a valid DEX file

After modification of its contents, a DEX file's header must be modified to be valid. The changes are located at the beginning of the header. First, there is a DEX magic value which should not be modified. Then, there are two fields: an Adler32 checksum and a SHA-1 hash of the file. Checksums are used to detect transmission or storage errors. Cryptographic hashes are used to ensure file's integrity. Both fields are computed over the rest of the file, i.e the SHA-1 hash is computed over the entire file excepted the DEX magic, the checksum and itself, and the checksum is computed over the entire file excepted the DEX magic and itself.

So, to re-build a valid DEX, we must do the computations in the following order:

1. Compute the DEX's SHA-1 and write it in the header
2. Compute the DEX's checksum and write it in the header. Note that the checksum includes data from the SHA-1 hash

Once this is done, the DEX is valid. Note the helper tool detailed in section 8 handles this step.

5 Re-building an APK

Once the DEX file is ready, we need to re-package it as an Android application (.apk). This step is not difficult for someone used to building Android applications in command lines, but may sound more obscure to Eclipse developers.

- Unzip the original APK and retrieve the compiled Android manifest (`AndroidManifest.xml`) and resources (strings in `resources.arsc`, layout, images...)

- Zip together the original manifest and resources with the new, modified, DEX file (`classes.dex`)
- Sign the package using `jarsigner` and developer keys

The process is particularly well suited to be automated in a Makefile (see an example of Makefile in Appendix).

6 Invoking a hidden method

Invoking a hidden method consists in the following steps:

1. **Open the DEX file of the current application** and put it in an array in memory. This is done using a method named `openNonAsset` of `android.content.res.AssetManager`. This method opens non-asset files as assets. This method is not directly accessible, so Java reflection must be used [Str12].
2. **Unpatch the DEX file array**. Basically, the idea is to undo what was done to hide the method at first (see section 3). The original values are restored.
3. **Open the unpatched DEX array as a new class**. This is done by calling, via reflection, `openDexFile()` on the DEX byte array. The method returns a ”cookie”, which is actually a pointer to an internal data structure for the DEX. Then, load the unpatched class (corresponding to the previously unpatched bytes) using `defineClass()`. This method uses the cookie returned by `openDexFile()` to load the class.
4. **Search for the hidden method** in the unpatched DEX file with `getDeclaredMethods()`)
5. **Invoke the hidden method** on an object instance from the unpatched DEX file

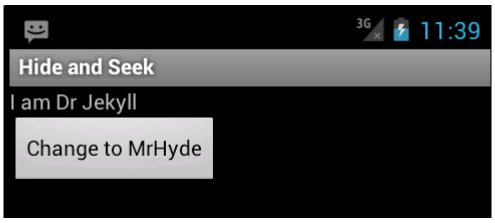
Example source code for this is provided in Appendix. A few logs in the code confirms the hidden method is called:

```
$ adb logcat
...
I/HideAndSeek(  644): invoking thisishidden() with arg=true
I/HideAndSeek(  644): In thisishidden(): set mrhyde=true
```

There is nonetheless an important limitation to understand when calling the hidden method. For the operating system, technically speaking, *the classes and methods of the original DEX are different from the classes and methods of the patched DEX,* even though they share the same name and namespace. It is consequently impossible to have a hidden method modify instance or static fields of a class: the fields of the original class are not affected by changes in fields in the patched class and reciprocally. Fortunately, there are alternatives, such as using shared files.

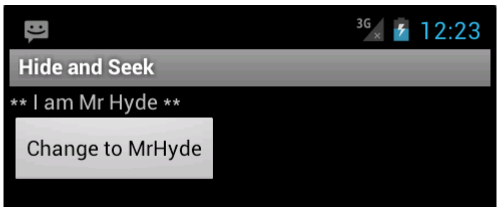
7 Proof of Concept application

A PoC application has been implemented and successfully tested on recent Android emulators (e.g 4.0.3, 4.1). It demonstrates how to invoke a hidden method.



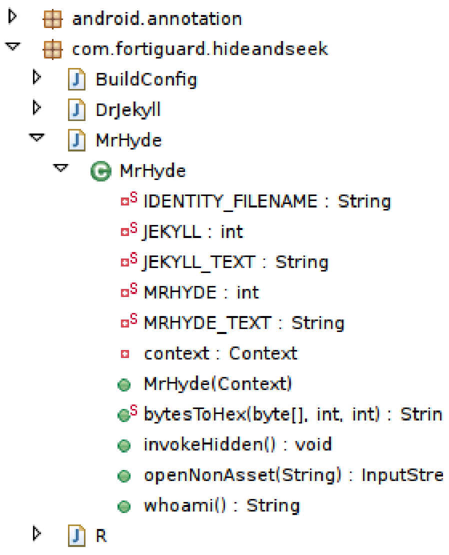
I am Dr Jekyll

Figure 3: Initially, Dr Jekyll has the right personality



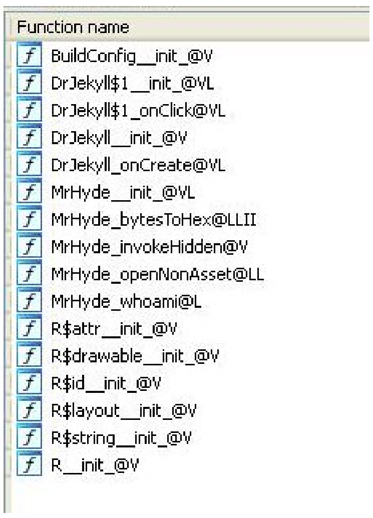
** I am Mr Hyde **

Figure 4: When the button is pressed - and the hidden method is called - Dr Jekyll changes into the evil personality of Mr Hyde



```
android.annotation
com.fortiguard.hideandseek
  BuildConfig
  DrJekyll
  MrHyde
    MrHyde
      IDENTITY_FILENAME : String
      JEKYLL : int
      JEKYLL_TEXT : String
      MRHYDE : int
      MRHYDE_TEXT : String
      context : Context
      MrHyde(Context)
      bytesToHex(byte[], int, int) : Strin
      invokeHidden() : void
      openNonAsset(String) : InputStre
      whoami() : String
  R
```

Figure 5: The hidden method, `thisishidden()`, does not show to dex2jar 0.0.9.13



```
Function name
BuildConfig__init_@V
DrJekyll$1__init_@VL
DrJekyll$1_onClick@VL
DrJekyll__init_@V
DrJekyll_onCreate@VL
MrHyde__init_@VL
MrHyde_bytesToHex@LLII
MrHyde_invokeHidden@V
MrHyde_openNonAsset@LL
MrHyde_whoami@L
R$attr__init_@V
R$drawable__init_@V
R$id__init_@V
R$layout__init_@V
R$string__init_@V
R__init_@V
```

Figure 6: The hidden method does not show either to IDA Pro 6.3

7.1 Demonstration

The demo application is made of two classes, DrJekyll and MrHyde, which are characters of the famous novel ”The strange case of Dr Jekyll and Mr Hyde” [Rob86]. In this novel, Dr Jekyll suffers, without being aware of it, from split personalities, and from time to time, he changes into the evil personality of Mr Hyde, Then he’s back to the ’normal’ Dr Jekyll.

The demo application is a virtual representation of Dr Jekyll. At first, it starts with the identity of Dr Jekyll. When the end-user presses the personality changing button, it (virtually!) changes into Mr Hyde’s identity. However, changing to Mr Hyde’s identity is only possible by the call of a hidden method (in our case, the hidden method is named `thisishidden()`). There is no other way to switch to Mr Hyde’s identity (not cheating!). If the application displays ”** I am Mr Hyde ***”, this means the call has been successful.

So, basically, we are going to switch to MrHyde’s identity using a hidden method that does not show to any Android disassembler: see Figures 5 and 6.

The hidden method does not show either to Androguard 1.7 using method name completion.

```
In [2]: d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.ME
```

```
d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.METHOD_bytesToHex
d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.METHOD_init
d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.METHOD_invokeHidden
d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.METHOD_openNonAsset_Ljava_lang_StringLjava_io_
d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.METHOD_whoami
```

If we list methods via interactive Python commands, a meticulous analyst might notice two methods are listed for `openNonAsset()`:

```
In [10]: methods = d.CLASS_Lcom_fortiguard_hideandseek_MrHyde.get_methods()
In [11]: for method in methods:
     print method.get_name();
    ....:
<init>
bytesToHex
invokeHidden
openNonAsset
openNonAsset
whoami
```

Same for Baksmali 1.4.2 which detects a *"method with a duplicate signature"*. This is because our patch assigns to the hidden method the same code offset as the previous method, which is `openNonAsset()` (see Table 1).

```
#Ignoring method with duplicate signature
#.method public openNonAsset(Ljava/lang/String;)Ljava/io/InputStream;
...
```

Nevertheless, neither Baksmali nor Androguard show the name or the code for the hidden method.

7.2 Implementation details of the application

The implementation of class `DrJekyll` is short:

- Displaying the view
- Retrieving the current personality: this corresponds to the `whoami()` method on a MrHyde object instance (see code lines 5-6 in 7)
- Calling the hidden method, `thisishidden`, when the button is pressed (see lines 9-13). Actually, `thisishidden` is called indirectly: pressing the button calls a method named `invokeHidden()` of `MrHyde`. This method is then in charge of the tricky part which consists in calling `thisishidden()`

The `MrHyde` class is more interesting. First of all, the current personality of Dr Jekyll / Mr Hyde is stored in a fixed file on the device. This file is read using `whoami()`, and written by calling the hidden method `thisishidden()`. **The code uses no other way to modify the current personality.** In particular, we are *not* accessing the identity file through another application, command line scripts nor hacking Dr Jekyll's TextView to display another string. Also, note that a non-existant identity file (which is the default initial situation) or invalid values map to Dr Jekyll's identity. So, changing into Mr Hyde requires an explicit call to `thisishidden()` where the boolean argument `ismrhyde` is explicitly set to true.

For reminder, the reason for reading the current identity through a file is explained in section 6: a static field would not have worked because there would have been two static fields: one for the original class and one

```java
public class DrJekyll extends Activity {
    ...
    public void onCreate(Bundle savedInstanceState) {
     ...
        hyde = new MrHyde(this.getApplicationContext());
        txtView.setText(hyde.whoami());

        validateBtn.setOnClickListener(new View.OnClickListener()  {
            public void onClick(View v) {
                Log.i("HideAndSeek", "DrJekyll: calling invokeHidden");
                hyde.invokeHidden();
                txtView.setText(hyde.whoami());
                Log.i("HideAndSeek", "DrJekyll: onClick done");
            }
        });
    }
}
```

Figure 7: The implementation of DrJekyll's class is very simple

for the patched one. With a slightly different approach to implementing Dr Jekyll's split identities, it would however have been possible not to use files. For example, the identity could have been stored as an instance field of DrJekyll class, and returned by the invokeHidden() method. The implementation in this paper is just one way of doing it.

How to load a hidden method has been convered by section 6. We follow those steps, sample code is provided in Appendix. We open the DEX file of the application, patch the DEX, list the methods and find the hidden method. We cannot invoke the hidden method, which belongs to the patched MrHyde class onto an object of the non-patched class: Android forbids it. Therefore, we must instantiate a new object for the patched class. Finally, we invoke the hidden method on the new object, using Java reflection, and provide the identity for `MRHYDE` as argument.

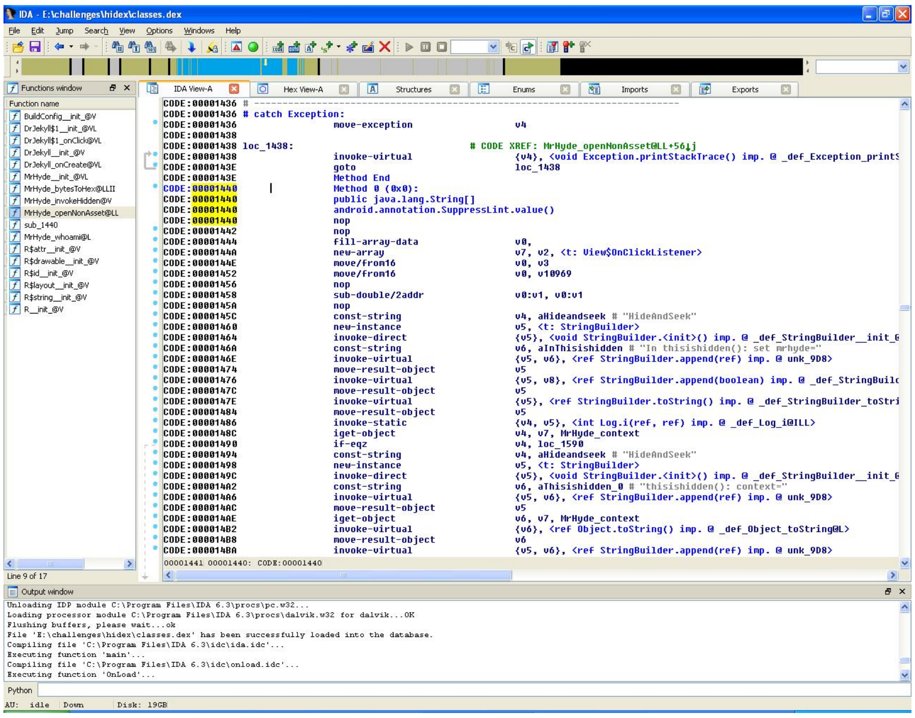
8 Patching/Restoring tool

I wrote a Perl script to automate the hiding or revealing of methods in DEX files. This script will be made available after the talk. It offers three functionalities:

1. Viewing the layout of the DEX file (without modification). This is useful to find and decide the offsets and values to patch.

```
./hidex.pl --input classes.dex --class "Lcom/fortiguard/hideandseek/MrHyde;"
...
 direct methods:
   method #0- name=<init> (0x13) (position=0x2C87)
     access      = 0x10001 (65537)
     code_offset= 0xD74 (3444)
     idx_diff    = 19
...
```

2. Patching the description of a method. The end-user provides the new values for each field.



```
CODE:00001436 # ---------------------------------------------------------------
CODE:00001436 # catch Exception:
CODE:00001436                 move-exception                  v4
CODE:00001438
CODE:00001438 loc_1438:                               # CODE XREF: MrHyde_openNonAsset@LL+56↓j
CODE:00001438                 invoke-virtual                  {v4}, <void Exception.printStackTrace() imp. @ _def_Exception_printS
CODE:0000143E                 goto                            loc_1438
CODE:0000143E                 Method End
CODE:00001440                 Method 0 (0x0):
CODE:00001440                 public java.lang.String[]
CODE:00001440                 android.annotation.SuppressLint.value()
CODE:00001440                 nop
CODE:00001442                 nop
CODE:00001444                 fill-array-data                 v0,
CODE:0000144A                 new-array                       v7, v2, <t: View$OnClickListener>
CODE:0000144E                 move/from16                     v0, v3
CODE:00001452                 move/from16                     v0, v10969
CODE:00001456                 nop
CODE:00001458                 sub-double/2addr                v0:v1, v0:v1
CODE:0000145A                 nop
CODE:0000145C                 const-string                    v4, aHideandseek # "HideAndSeek"
CODE:00001460                 new-instance                    v5, <t: StringBuilder>
CODE:00001464                 invoke-direct                   {v5}, <void StringBuilder.<init>() imp. @ _def_StringBuilder__init_@
CODE:0000146A                 const-string                    v6, aInThisishidden # "In thisishidden(): set mrhyde="
CODE:0000146E                 invoke-virtual                  {v5, v6}, <ref StringBuilder.append(ref) imp. @ unk_9D8>
CODE:00001474                 move-result-object              v5
CODE:00001476                 invoke-virtual                  {v5, v8}, <ref StringBuilder.append(boolean) imp. @ _def_StringBuilc
CODE:0000147C                 move-result-object              v5
CODE:0000147E                 invoke-virtual                  {v5}, <ref StringBuilder.toString() imp. @ _def_StringBuilder_toStri
CODE:00001484                 move-result-object              v5
CODE:00001486                 invoke-static                   {v4, v5}, <int Log.i(ref, ref) imp. @ _def_Log_i@ILL>
CODE:0000148C                 iget-object                     v4, v7, MrHyde_context
CODE:00001490                 if-eqz                          v4, loc_1590
CODE:00001494                 const-string                    v4, aHideandseek # "HideAndSeek"
CODE:00001498                 new-instance                    v5, <t: StringBuilder>
CODE:0000149C                 invoke-direct                   {v5}, <void StringBuilder.<init>() imp. @ _def_StringBuilder__init_@
CODE:000014A2                 const-string                    v6, aThisishidden_0 # "thisishidden(): context="
CODE:000014A6                 invoke-virtual                  {v5, v6}, <ref StringBuilder.append(ref) imp. @ unk_9D8>
CODE:000014AC                 move-result-object              v5
CODE:000014AE                 iget-object                     v6, v7, MrHyde_context
CODE:000014B2                 invoke-virtual                  {v6}, <ref Object.toString() imp. @ _def_Object_toString@L>
CODE:000014B8                 move-result-object              v6
CODE:000014BA                 invoke-virtual                  {v5, v6}, <ref StringBuilder.append(ref) imp. @ unk_9D8>
00001441 00001440: CODE:00001440
```

```
Unloading IDP module C:\Program Files\IDA 6.3\procs\pc.w32...
Loading processor module C:\Program Files\IDA 6.3\procs\dalvik.w32 for dalvik...OK
Flushing buffers, please wait...ok
File 'E:\challenges\hidex\classes.dex' has been successfully loaded into the database.
Compiling file 'C:\Program Files\IDA 6.3\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA 6.3\idc\onload.idc'...
Executing function 'OnLoad'...
```

Figure 8: Declare a new function at the hidden method's location to reveal the hidden code

```
./hidex.pl --input classes.dex --patch-offset 0x2c99 --patch-idx 0
  --patch-flag 1 --patch-code 0x13d8 --patch-next-idx 2
```

9 Solutions: discussion

Hiding a method in a DEX file can be used in several ways, such as bypassing market places' screening or as an anti-reversing technique. In the case of malware, anti-virus analysts are fortunately not helpless. They can spot anomalies such as strings present in the DEX but not used (those are strings used by the hidden method) or duplicate method signature after disassembly (see Section 7).

Once the analyst suspects a hidden method, he/she can reveal it using one of the following methods:

- **IDA Pro**. Inspect each method and look for undefined data at the end. Try to declare that data as a new function (shortcut "P" in IDA Pro of the first undefined byte). When placed at the right location, IDA Pro will treat the following byte-code as valid Dalvik instructions and reveal the hidden method. See Figure 8.

- **Androguard**. I wrote a Python script using Androguard that disassembles a DEX at any offset. The tool is named androdis, and has now been added to Androguard's repository [Apv12a]. Use it at the exact location where the hidden method should be to reveal the hidden instructions. Alternatively, those actions can also be performed from Androguard's interactive shell (`androlyze -s`).

```
$ ./androdis.py -i classes.dex -o 5196
0 0x0 move-object/16 v2, v3
```

```
0x6 move/from16 v0, v10969
0xa nop
0xc sub-double/2addr v0, v0
0xe nop
0x10 const-string v4, 'HideAndSeek'
0x14 new-instance v5, Ljava/lang/StringBuilder;
0x18 invoke-direct v5, Ljava/lang/StringBuilder;-><init>()V
0x1e const-string v6, 'In thisishidden(): set mrhyde='
..
```

- **Unpatch** the DEX file, and basically undo what was done to hide the method. Then re-sign, checksum the DEX and re-build the APK (section 5).

From an OS point of view, Android could be patched to verify the consistency of encoded_method fields, like it is done for string identifiers. In particular, the OS could verify each encoded_method entry references a different method signature and offset. Google has been notified on June 24th (Ref. 1314462818).

10 Conclusion

This paper exploits a lack of format verification in Dalvik Executables to hide, at will, a method from decompilers. The hidden method remains in the DEX file, it is just invisible. With an appropriate piece of code, it can even be called and get executed. A Proof of Concept application demonstrating this has been implemented, along with a tool able to patch DEX files.

From a defensive aspect, the paper also provides 3 different ways to spot hidden methods. Alternatively, the Android OS could be patched to include verifications.

11 Additional information

- Presenter: Axelle Apvrille, France
- Employer: Fortinet
- Brief biography:

  > Axelle Apvrille's is currently a Senior Antivirus analyst and researcher at Fortinet, where she more specifically looks into mobile malware, and exercises her security blogger talents. On mobile malware, she is best known for her reverse engineering of Symbian/Yxes and Zitmo, the mobile add-on to ZeuS botnets.
  >
  > Axelle presented at various conferences, including BlackHat, RSA, ShmooCon, VB, EICAR (best paper award)...
  >
  > Known in the community by her more or less mysterious handle ”Crypto Girl”, she changes from office worker during the day into mighty hacker at night. Like Neo, but with a superhero costume.

- List of publications and papers: `http://wikisec.free.fr/papers/papers.html`
- Presentation or educational experience: see bio. Presented a several conferences, I've also been a lecturer in several French engineering schools.

- Reason why this material is innovative or significant: as far as I know, this is pioneer work regarding steganography within Dalvik Executables. Moreover, I believe that the mix between technical explanations and demos in this talk should be attractive to attendees.
- Prepared material: this paper, see code in appendix.
- Already presented: no, this has not been presented yet.

Acknowledgements

I ought to thank Pau Oliva (@pof) who worked on scripting a command to hide any executable. He was enthousiastic about this idea of hiding methods in a DEX file, and I thank him for pushing the idea forward... and keeping my motivation up :)

Appendix: source code

Example of Makefile to build an APK with a modified DEX

See code at Figure 9

Example source code for invoking a hidden method

See code below.

```
ROOTDIR = /foo/bar
PROJECTDIR = $(ROOTDIR)/app
MANUFACTUREDIR = $(ROOTDIR)/re-manufactured
ZIP = zip
JARSIGNER = jarsigner
JARSIGNER_OPTIONS = -verbose
KEYSTORE = $(PROJECTDIR)/yourkeystore
STOREPASS = theunsecurepassword
ALIAS = youralias

all: release.apk

release.apk: release-unsigned.apk
        $(JARSIGNER) $(JARSIGNER_OPTIONS) -keystore $(KEYSTORE)
       -storepass $(STOREPASS) -signedjar release.apk
       release-unsigned.apk $(ALIAS)

release-unsigned.apk: resources.arsc manifest classes.dex logo.png main.xml
        $(ZIP) release-unsigned.apk AndroidManifest.xml resources.arsc
     classes.dex res/*/logo.png res/layout/main.xml

manifest: $(PROJECTDIR)/bin/release-unsigned.apk
        unzip $(PROJECTDIR)/bin/release-unsigned.apk AndroidManifest.xml

# same for main.xml, resources.arsc etc

clean:
        rm -f release.apk
        rm -f release-unsigned.apk
        rm -f AndroidManifest.xml
        rm -f resources.arsc
        rm -rf res/
        rm -f *~
```

Figure 9: Makefile for re-building an APK

```
    public void invokeHidden() {
        int dexlength = 11980;
        byte[] dex = new byte[dexlength];

        // load this DEX file
        InputStream localInputStream = openNonAsset("classes.dex");
        try {
            dexlength = localInputStream.read(dex, 0, dexlength);
            Log.i("HideAndSeek", "invokeHidden(): read "+dexlength + " bytes");
            // modify dex here and un-hide hidden method
            int patch_index = 0x2da9;
            dex[patch_index++]= 1; // method_idx
            dex[patch_index++]= 1; // access flag
            dex[patch_index++]= (byte)0xbc; // code offset
            dex[patch_index++]= (byte)0x29;
            dex[patch_index++]= 1;

          // re-compute the checksum and hash
            MessageDigest digest;
            digest = MessageDigest.getInstance("SHA-1");
            digest.reset();
            digest.update(dex, 32, dexlength-32);
            digest.digest(dex, 12, 20);
            Log.i("HideAndSeek", "invokeHidden(): redigesting");

            Adler32 checksum = new Adler32();
            checksum.reset();
            checksum.update(dex, 12, dexlength-12);
            int sum = (int)checksum.getValue();
            dex[8] = (byte)sum;
            dex[9] = (byte)(sum >> 8);
            dex[10] = (byte)(sum >> 16);
            dex[11] = (byte)(sum >> 24);
            Log.i("HideAndSeek", "invokeHidden(): checksum");

            // get DexFile class object
            // search for openDexFile and defineClass methods
            Class dexFileClass = context.getClassLoader().loadClass("dalvik.system.DexFile");
            Method[] arrayOfMethod = Class.forName("dalvik.system.DexFile").getDeclaredMethods();
            Method openDexFileMethod = null;
            Method defineClassMethod = null;
            int cookie = 0;
            Log.i("HideAndSeek", "invokeHidden(): openDexFile");

            for (int i=0; i< arrayOfMethod.length; i++) {
                if (arrayOfMethod[i].getName().equalsIgnoreCase("openDexFile")
&& arrayOfMethod[i].getParameterTypes().length == 1) {
                    openDexFileMethod = arrayOfMethod[i];
                    openDexFileMethod.setAccessible(true);
                    Log.i("HideAndSeek", "openDexFile found");
                }
                if (arrayOfMethod[i].getName().equalsIgnoreCase("defineClass")
&& arrayOfMethod[i].getParameterTypes().length == 3) {
                    defineClassMethod = arrayOfMethod[i];
                    defineClassMethod.setAccessible(true);
                    Log.i("HideAndSeek", "defineClass found");
                }
            }

            // call openDexFile()
            Object[] arrayOfObject = new Object[1];
            arrayOfObject[0] = dex;
            Log.i("HideAndSeek", "dex header: "+MrHyde.bytesToHex(dex,0, 34));

            if (openDexFileMethod != null) {
                cookie = ((Integer)openDexFileMethod.invoke(dexFileClass, arrayOfObject)).intValue();
                Log.i("HideAndSeek", "openDexFile invoked. cookie="+cookie);
            }
```

```
           // call defineClass -> get an object for the new MrHyde class
            // invoke: private native static Class defineClass(String name, ClassLoader loader, int cookie);
            Object[] params = new Object[3];
            params[0] = "com/fortiguard/hideandseek/MrHyde";
            params[1] = dexFileClass.getClassLoader();
            params[2] = Integer.valueOf(cookie);
            Class patchedHyde = null;
            Log.i("HideAndSeek", "retrieving patched MrHyde class");
            if (defineClassMethod != null) {
                patchedHyde = (Class) defineClassMethod.invoke(dexFileClass, params);
            }

            // search for thisishidden in the new class object
            // invoke:   public void thisishidden()
            Method thisishiddenMethod = null;
            Log.i("HideAndSeek", "getting methods in patched MrHyde");
            Method[] allMethods = patchedHyde.getDeclaredMethods();
            Log.i("HideAndSeek", "parsing methods in patched MrHyde:");

            for (int j=0; j<allMethods.length; j++) {
                Log.i("HideAndSeek", "patched MrHyde method: "+allMethods[j].getName());
                if (allMethods[j].getName().equalsIgnoreCase("thisishidden")) {
                    thisishiddenMethod = allMethods[j];
                    Log.i("HideAndSeek", "thisishidden() method has been found");
                }
            }
            Log.i("HideAndSeek", "parsing done.");

            // invoke: public void thisishidden(boolean);
            if (thisishiddenMethod != null) {
                Object[] arg= new Object[1];
                Log.i("HideAndSeek", "before new Instance()");
                // instantiate an object of the new MrHyde class
                Object obj = patchedHyde.getDeclaredConstructor(Context.class).newInstance(context);
                Log.i("HideAndSeek", "after new Instance");
              // call thisishidden on that object instance
                arg[0] = Boolean.valueOf(true);
                Log.i("HideAndSeek", "invoking thisishidden() with arg=true");
                thisishiddenMethod.invoke(obj, arg);
            } else {
                Log.i("HideAndSeek", "thisishidden() not found");
            }
        }
        catch(Exception exp) {
            Log.e("HideAndSeek", "Exception caught in invokeHidden(): "+exp.toString());
        }
    }
}
```

References

[AOSa] Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html.

[AOSb] .dex — Dalvik Executable Format. http://source.android.com/tech/dalvik/dex-format.html.

[Apv12a] Axelle Apvrille. Androdis source code, 2012. https://code.google.com/p/androguard/source/browse/androdis.py.

[Apv12b] Axelle Apvrille. Guns and Smoke to Defeat Mobile Malware, November 2012. Hashdays Conference, Lucerne, Switzerland.

[Rob86] Robert Louis Stevenson. *Strange Case of Dr Jekyll and Mr Hyde*. Longmans, Green and co, 1886.

[Str12] Tim Strazzere. Dex Education: Practising Safe Dex. In *BlackHat USA*, July 2012.