Packadroid
A Framework for Repackaging Android Applications

Marko Dorfhuber (03658730) ✉marko.dorfhuber@tum.de
Max Hornung (03662676) ✉maximilian.hornung@tum.de
Moritz Oettle (03658085) ✉moritz.oettle@tum.de

1. Contribution

Repackaging of *Android* applications is a serious problem, since it embeds malware into existing apps and may defrauds the user. In addition, this allows to create a lot of different applications that all perform the same malicious actions. As the analysis of malware is a process still done manually most of the time, the creation of a lot of different malicious applications is a serious problem.
In our class project, we introduce a novel perspective on repackaging malware. In the past, creating repackaged apps remained a lot of effort, e.g. manual rewriting of `Smali` files has been required. This hindered the understanding of repackaging mechanisms and research in this area. In addition, our approach of repackaging allows an automation which underlines the need for an automated malware analysis on the one hand. On the other hand, it could be utilized to create a lot of malware samples which can be used in research for machine learning-based detection mechanisms.
Because the modifications needed to embed additional software into an app are not technically complicated, we show how to automate them in our framework named **Packadroid** [9]. Our framework is able to inject arbitrary malicious payloads into arbitrary original applications. Furthermore the injected payloads are hooked into the existing original application. This means a user of the framework decides, on which occasion his malware is launched. This generates new challenges for security researches, because with our tool it is possible to perform repackaging of applications in a larger scale.

2. Background

In the *Android* OS, apps are deployed as `apk` files. This file type is a zip file that can be unpacked with common tools. Furthermore, there are tools like `apktool` which are able to unpack the application and decompile the `dex` bytecode to `Smali` source code. Additionally, `apktool` [1] is able to rebuild the application after modifications on the `Smali` code.
This allows users to change various features of the application, ranging from differently colored layouts to the injection of additional code. To modify the application, the decompiled `Smali` code has to be changed. `Smali` is a human readable disassembled intermediate language of *Dalvik* byte code, where *Dalvik* is the *Java* VM implementation of *Android* [8]. The `Smali` language is the 'assembler' code used by *Dalvik*, the *Java* VM implementation of *Android* [8]. One of the most crucial differences to *Java* is that the `dex` bytecode is register oriented, while the *Java* bytecode is stack-based.

To explain the fundamentals of this language, a `Smali` hello-world code snippet is given in Listing 2. In this listing, we can see that classes are marked by a preceding L, while the dots in the package name are converted to slashes [8]. For example, the class `java.lang.String` is represented as `Ljava/lang/String;`. The return types are specified at the end of a method declaration, where 'V' indicates a void method. Last but not least, the "[" declares an array of the object class specified behind this bracket. Therefore, Line 5 of the code snippet declares the method with the *Java* signature '`public static void main(String[] args)`'.
In Line 6, the number registers utilized in the method is set to 2 using the `.registers` directive. This part must be specified in the header of each method. It is crucial to provide the number of registers used in a method to keep the Smali code correct. Therefore, this number may have to be changed when introducing new code into an existing method. After that, the `printstream` object of `java.lang.System` is loaded into the register v0. This object corresponds to typing `System.out` in *Java*. In Line 10, a reference to the string constant "Hello World!" is put into the register v1 using the `const-string` command. The `invoke-virtual` command is used to call the `println()` method of the object in register v0 with the argument given in register v1.
In order to call other methods in `Smali`, invoke statements are used. In order to call a static command, the `invoke-static` has to be used, as seen in Listing 1. If the method of a parent class should be called, then `invoke-super` should be used. It is important that `invoke-super` uses the vtable of the parent class to decide which method to call, while `invoke-virtual` uses the vtable of the target class [7].
Therefore, the actual type of the target object only matters to `invoke-vitual` and `invoke-super` only refers to its parent class type. If the method of a class should be called directly (without considering vtables), then `invoke-direct` can be used.
This hello-world example introduces all commands necessary to understand our approach. An overview about all possible commands is given in [3].

```
invoke-static {p0}, Lcom/metasploit/stage/Payload;->start(Landroid/content/Context;)V
```

Listing 1: Starting the payload code in `smali`

```
1  .class public LHelloWorld;
2
3  .super Ljava/lang/Object;
4
5  .method public static main([Ljava/lang/String;)V
6      .registers 2
7
8      sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
9
10     const-string  v1, "Hello World!"
11
12     invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
13
14     return-void
15 .end method
```

Listing 2: Hello world in `smali`, adapted from [4]

Since the register based `Smali` is cumbersome to develop and the merging of different `Smali` components is laborious, it is desirable to create a tool that automatically injects code into applications in a robust way.

3. Explanation of Approach

In this section, we will explain the design of our automated repackaging framework, as shown in Figure 1. At first `apktool` is used to decompile the original application, as shown in Listing 3. Hereafter, arbitrary malware payloads can be injected into the decompiled application. Furthermore, arbitrary hooks can be specified which invoke the malware payloads. All these modifications are performed on the `Smali` code of the original application. At last, `apktool` is used to rebuild the application and `jarsigner` [5] is utilized for signing the new malicious application. It is necessary to sign applications on *Android*. Both the `Play Store` and the application installer on the device will refuse installation of unsigned apps [2]. Therefore, the patched apps are signed with `jarsigner`.

```python
def decompileApk(apkPath):
    if not os.path.isfile(apkPath):
        raise Exception("Cannot find apk file at path {}".format(apkPath))
    outDir = os.path.join(os.path.splitext(apkPath)[0], "_decompiled")
    os.system("apktool d -o {} {}".format(outDir, apkPath))
    return outDir
```

Listing 3: Decompilation using `apktool` in python

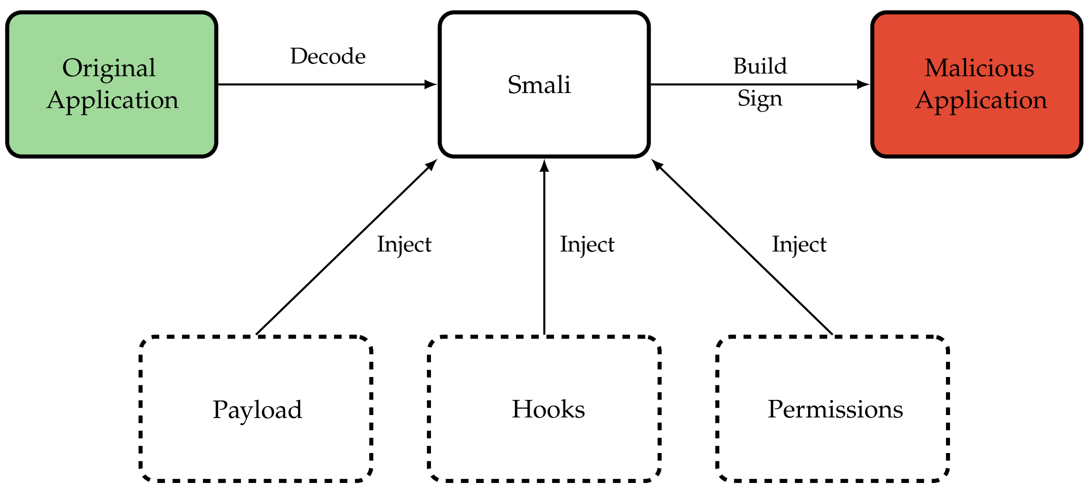

Figure 1: Process of repackaging

In the following, we describe an example workflow of our tool that injects source code into the *WhatsApp* application. First, the user launches **Packadroid** and types `help` to get an overview of available commands. When the user has read the commands, he loads the original *WhatsApp* apk file using `load_original` command. To understand the options for placing his own hooks, the user enters `list_activities`, which shows him all available activities from the original app. In order to ease the handling of apps with a large number of activities, we provide an activity ID to uniquely identify each activity. This feature can be used by the user, when he adds a hook to an activity using the `add_activity_hook` command. Instead of an activity name, this command also accepts the activity ID of the desired activity.
Another way to launch the malicious payload is the `add_broadcast_hook` command. This command enables the user to specify on which action (e.g. an incoming SMS or the beginning of charging the phone) the malicious code should be launched. We add an broadcast receiver to

the original apk file, and specify which of the possible actions should trigger the execution of the Receiver. The `Smali` code for the broadcast receiver is generated dynamically based on the specified launchers. A full list of possible broadcast actions is given in Table 1.

Table 1: Overview of available broadcast hooks.

Name of Broadcast	Additional information
on_power_connected	This hook will be executed once the user of the repackaged app starts to charge his phone. Since the battery capacity of modern smartphones is quite limited, it is highly probable for this hook to be executed.
on_power_disconnected	Once the user unplugs his phone from the charging cable, this kind of hook will be executed. Assuming that the user will likely not use his phone while it is loading, this hook could be used in combination with the previous one to execute malicious parts of the repackaged app unnoticed from the user.
on_boot_completed	After the phone is booted, this kind of hook is executed. This enables the designer of the repackaged app to execute code before the original app was started.
on_receive_sms	When a SMS is sent to the with the repackaged app installed, hooks of this kind are executed. Therefore, these hooks make the phone controllable by an external attacker, i.e. the attacker can trigger certain actions on the phone by sending specific SMS messages. Note that the repackaged app using this hook will always be able to read SMS, since this permission is required to monitor incoming SMS messages.
on_incoming_call	Similar to the on_receive_sms hooks, these kind of hooks are executed once the phone with the repackaged app is called. Again, this gives possible attackers the possibility to control the victim device. However, it is more hard to send specific messages because the audio signal can not be read as easily as a SMS message.
on_outgoing_call	After the user calls another person, this kind of hook is executed. Note that this class of hooks requires the `PROCESS_OUTGOING_CALLS` permission which enables the designer of the repackaged app to redirect or even abort phone calls of the user.

To make the repackager able to inject the payload code in the original app, both the activity based and the broadcast based hooks add the launching `Smali` code to the `smali/` folder of the original apk file.

After the `Smali` code injection is complete, the manifest of the original app is updated in order to conform to the newly added code. Several hooks need additional permissions, e.g. the "on_boot_completed" hook needs the `RECEIVE_BOOT_COMPLETED` permission. These missing permissions are now added to the manifest in order to keep the app working with the injected payload. Furthermore `<receiver>` have to be added for broadcast hooks. These receiver allow the broadcast receiver to catch the specified intents.

In the end, the user can repack the changed apk file to a desired location, which includes all specified payloads and hooks. The repackaging process combines the old apk file with the additional `Smali` code and the adjusted `AndroidManifest.xml`, such that the payload is executed at the desired events.

4. Implementation

For the development of our automated repackaging framwork **Packadroid**, we chose *Python*, because this modern scripting language is widely used in practice. The general design of the framework is illustrated in Figure 2.

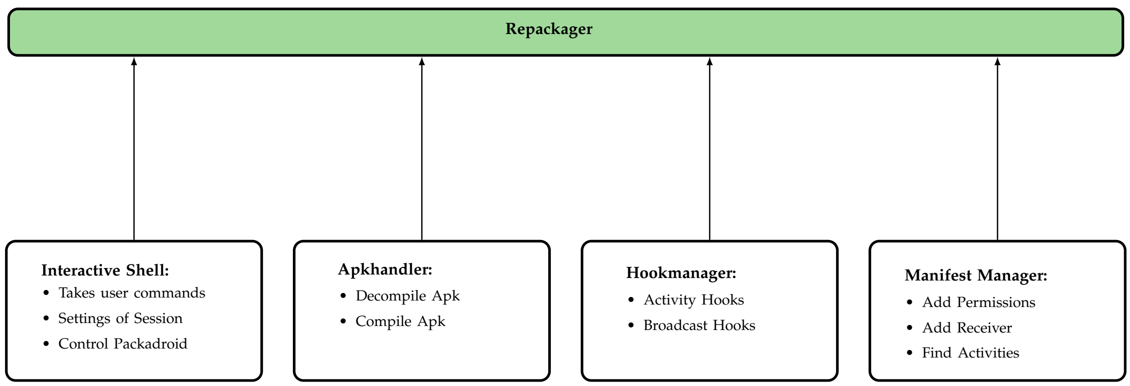

Figure 2: Implementation Overview

In order to launch `Packadroid`, the script *repackager.py* has to be executed. This script starts a shell, that was developed in our `interactive_shell` module. In this module, the `Prompt` class (based on the `Cmd` class provided by *Python*) provides the shell interactions. Additionally, the `PackadroidSession` class stores the current state (e.g. which hooks have already been specified) and works as a bridge to the other modules. To achieve this, it offers methods that provide certain functions, e.g. `load_original_apk` that loads the information of the application that shall be repackaged using various methods of the other modules.
It is also possible to start a *Metasploit* console and use an existing *Metasploit* function to generate the payload for a reverse shell. *Metasploit* [6] is an open source pentesting framework that includes various attacks.
In order to handle the task of actual repackaging, our tool provides the `apkhandling` module, that contains all methods necessary to decompile apk files and rebuild them after the payload has been injected. In this module, we also implemented the required signing of the repackaged apk files.
The `manifestmanager` module provides an interface to extract necessary information of the `AndroidManifest.xml`. For example, it is possible to retrieve the permissions granted to an app or the (launcher) activities of the original app. Furthermore, this module has the capability to change certain elements of the provided apk file, e.g. to add required permissions or broadcast receivers to the manifest file.
Last but not least, the `hookmanager` module handles the hook information and provides methods needed to integrate hooks into the repackaged apk file. The information about a certain hook is stored in `Hook` objects. Based on provided hook objects, the `activity_hook.py` enables the injection of `Smali` code to activate the desired hook. If the hook should be invoked if a certain intent is received instead of beeing activated if an activity is loaded, the `broadcast_hook.py` provides the corresponding methods. This module needs also access to the `AndroidManifest.xml`

file to add more permissions for broadcast hooks, since the original app may miss the capability of registering when a certain event happened. For example, if the original app does not have the `READ_SMS` permission, then the app will not be able to monitor when a SMS arrived and it would not be possible to trigger the hook.

5. Commands

Our repackaging tool is implemented with an interactive shell. We provide several commands that enable specific repackaging actions for the user. In the following, the commands are explained in detail.
All those commands can also be specified in a batch file to automate the repackaging. Therefore, we provide a command line parameter (`-b`). Using this parameter the user can specify a batch file that contains simple the commands he would type into the interactive shell. The framework will execute all commands of the file on startup.

- `help` – Display explanation for all commands on shell.
- `exit` – Closes the shell, no changes are applied.
- `add_activity_hook` – When a hook is added using this method, it will be executed once the specified activity is loaded. This command requires to specify the payload `.apk` file, together with the class to be used. Additionally, the payload method has to be provided. This is the method of the hook to be executed. The specified method must be specified public and static.
- `add_broadcast_hook` – Similar to `add_activity_hook`, this command adds a hook that is executed once a specified broadcast receiver receives a certain intent. For example, it is possible to execute payload code after the user starts to load the device.
- `load_original` – This command expects a path to an *Android* `.apk` file. If the path is valid, `apktool` is used to decompile the corresponding `.apk` file.
- `load_activities` – It is possible to load all activities specified in a given `AndroidManifest.xml` file. These activities may be used in `load_activity_hook`.
- `list_added_hooks` – Once a user has loaded one or more hooks, it may be cumbersome for the user to remember all of them. Therefore, we provide this method that shows all already added hooks.
- `remove_hook` – With this command, the user can remove an already loaded hook by providing its ID. Note that the IDs are in the interval from 0 to n-1, and can be shown using the `list_added_hooks`.
- `start_meterpreter_handler` – This command provides the user a commodity possibility to start *Metasploit*. This method should be called before `generate_meterpreter`. It requires to specify an IP and a port for the *Metasploit* instance to be started.
- `generate_meterpreter` – Creates the payload necessary to create a reverse shell. This command requires *Metasploit* to be installed! It requires to specify an IP and a port for the *Metasploit* instance to be used.

- `repack` – This command uses the specified hooks and merges the payload contents with the original ones. It should be executed after all other commands, because the repackaged app will only contain features specified before this method was called.

For running and installing the *Packadroid* framework, refer to the *Github* repository with the source code [9].

6. Extensibility

We implemented our framework **Packadroid** with a focus on extensibility, so it is possible to extend it further in various directions. It was more important to us to provide robust software framework than extending it too fast without the ability to guarantee that these also work.
For example, it is possible to add more events for the broadcast receiver without having to change the structure of the project. This would make it possible to inject code on other intents than the supported ones.
Additionally, we only utilize a small fraction of the *Metasploit* framework. This open source framework provides a huge number of additional payloads that are already implemented and just wait to be integrated into **Packadroid**, which is eased a lot by our project structure that already provides the functionality to execute specific *Metasploit* commands.

7. Future Work

The main part of future work for this project is the extension of the payload library. We provide a set of precompiled payloads, like sending an SMS or deleting all contacts. However, this list can be extended arbitrarily.
We provide the possibility to add activity-based and broadcast receiver-based hooks. Those two hooks show that the automatic repackaging of *Android* applications is possible. Nevertheless, there could be additional possibilies to add hooks, like general *Java* classes. The possibility of adding *Semantic Hooks* would be a great extension of the framework, too. *Semantic Hooks* are hooks that are for example triggered if the application reaches a special state of the user enters a special string into a text field. However, the framework has to understand the semantics of the program or the application's state to be able to inject such hooks. This is very complicated and requires a lot of additional research.
The framework could be extended to obfuscate the payload automatically while repackaging. This circumvents the easy analysis of the payload. Nevertheless, the payload can currently be obfuscated manually as long as the user specifies the correct entry point. In addition, the obfuscation of the whole application is still possible after repackaging.
The last extension we wanted to list for future work is the improvement of usability. This means that additional commands are added to the interactive shell, like giving a warning if the user specifies a payload entry point that is not available in the specified payload application. This is just an engineering task and does not affect our approach or the general operatability of the framework.
As we stated in Section 6, one focus of the implementation is extensibility. Therefore, it is possible to add the future work to the running implementation easily.

References

[1] Apktool – a tool for reverse engineering 3rd party, closed, binary android apps. https://ibotpeaches.github.io/Apktool/. Accessed: 02.02.2018.

[2] Application signing. https://source.android.com/security/apksigning/. Accessed: 27.01.2018.

[3] Dalvik opcodes. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html. Accessed: 27.01.2018.

[4] Hello world in smali. https://github.com/JesusFreke/smali/blob/master/examples/HelloWorld/HelloWorld.smali. Accessed: 27.01.2018.

[5] jarsigner. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html. Accessed: 27.01.2018.

[6] Metasploit, the worlds most used penetration testing framework. https://www.metasploit.com/. Accessed: 27.01.2018.

[7] .smali invoke-virtual & invoke-super. https://stackoverflow.com/questions/23521442/smali-invoke-virtual-invoke-super. Accessed: 27.01.2018.

[8] smali/baksmali github repository. https://github.com/JesusFreke/smali. Accessed: 27.01.2018.

[9] M. Dorfhuber, M. Oettle, and M. Hornung. Packadroid. https://github.com/PraMiD/Packadroid. Accessed: 06.02.2018.