PACKING AND UNPACKING ANDROID APPLICATIONS

Bitdefender

Dynamically load code

Android allows dynamic code loading using standard APIs

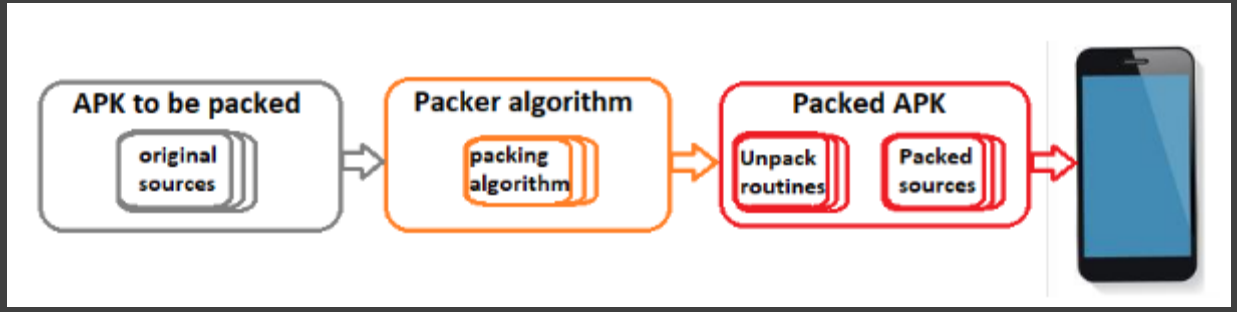
- load classes.dex files Some examples APIs
  - dalvik.system.DexFile.loadDex (deprecated in API 26)
  - dalvik.system.DexClassLoader
  - dalvik.system.PathClassLoader
- load .so (native executable) files
  - Java.lang.System.loadLibrary
  - Java.lang.System.load

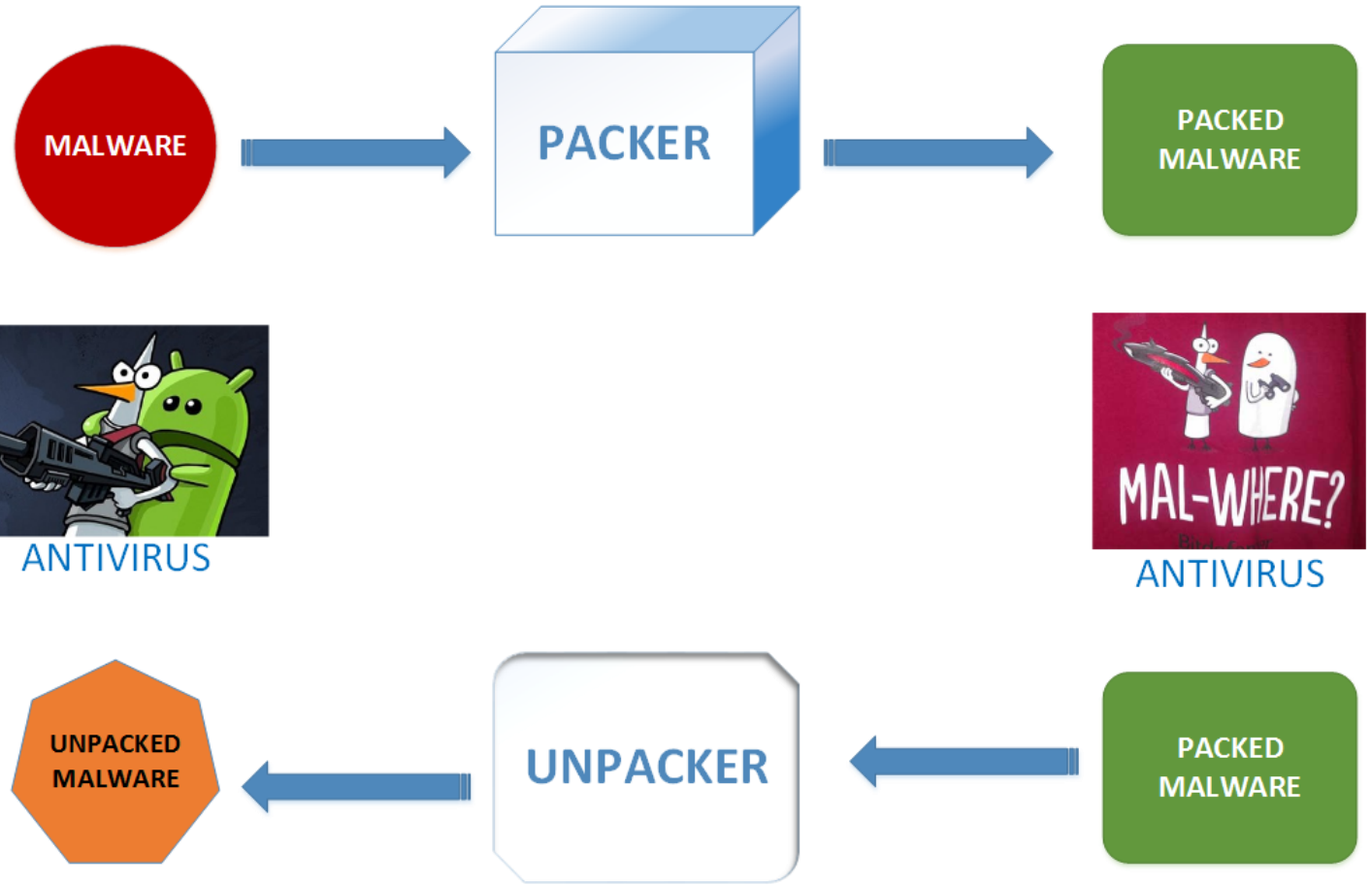
Dynamic code can also be loaded by:

- using custom loaders (implementing the entire loading process using in-app code, not predefined APIs). Some packers use this
- Using a different code format/mechanism. Example reading .js files and executing the JavaScript code via loadUrl

Packing

- A packer is a mechanism that wrapps the given APK, mentaining its original functionality while hiding the app code and only retrieving it at runtime.
  - Example of packing: encrypt the original classes.dex file, decrypt the DEX file to memory at runtime, and then execute via DexClassLoader.
- Packers assure that
  - intellectual property is protected (original purpose of packers)
  - Copycats can not directly get the source code (think jadx – decompilation)
- Packing is not bulet-proof, given a generous amount of time, any unpacking can be done



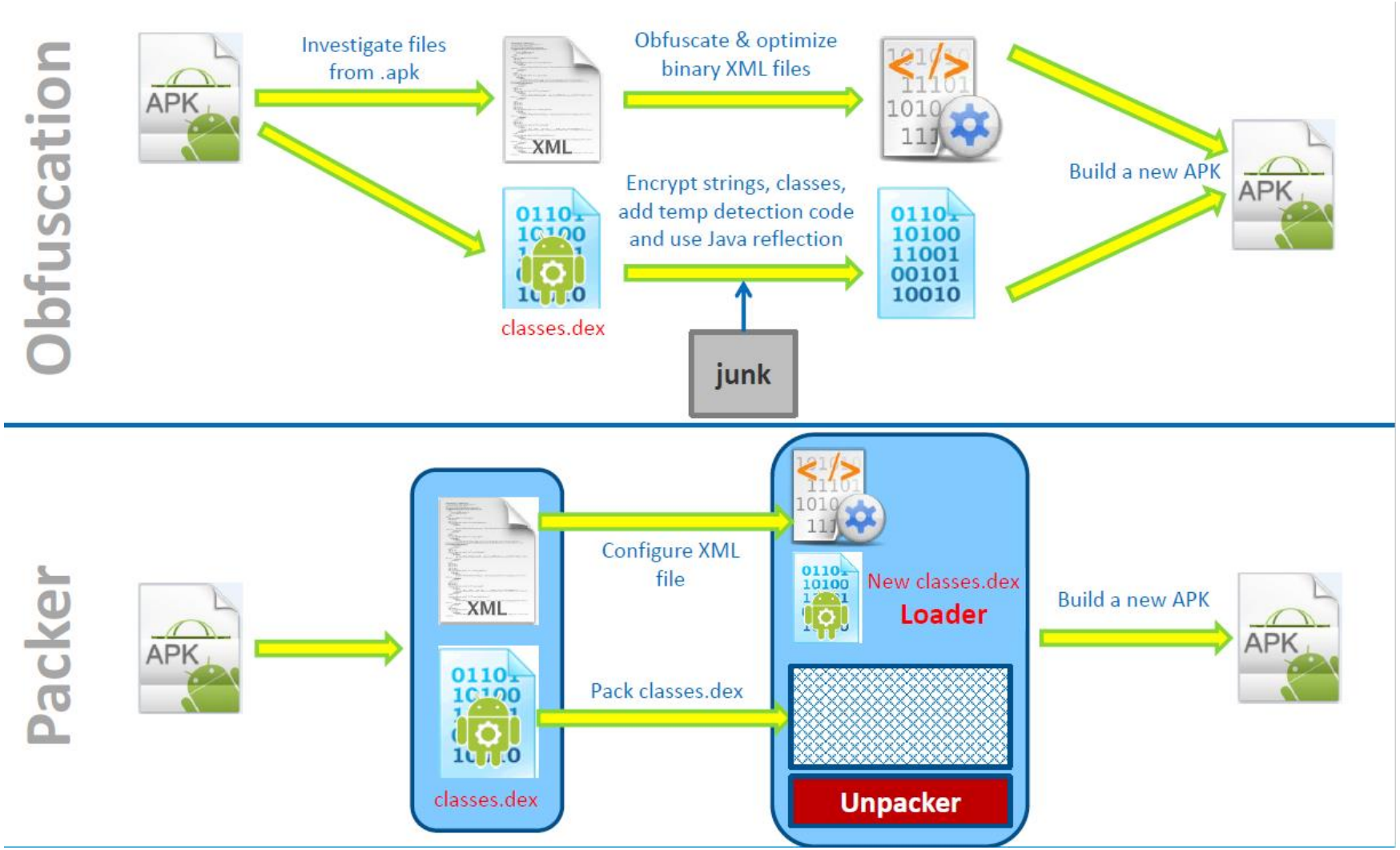


Packing

- Malware use Packers to hide code from automatic security solutions' scanners.
- To counter this, unpacking is done/implemented

Packing is not Obfuscation

Packing – hiding code; Obfuscation: modifying code making it unreadable



Packer examples

Legu

- Unpacking routine in libshell*.so
- Encrypted files and metadata in the final APK's assets
- An analysis can be found here

Bangcle

- Compresses and encrypts all files
- Decrypts, decompresses and loads in unpacking routine
- An analysis can be found here

Ijiami

- Libexec.so: verifies integrity of encrypted data then decrypts it
- Libexecmain.so: runs libexec.so and loads original code
- An analysis on Ijiamy and other can be found here

Employed Mechanisms

- Anti-Temper
  - Checks integrity of the packed data (e.g. signature SHA1)
- Anti-Decompiler
  - Load original DEX at runtime
- Anti-Debug
  - Check for emulator/debugger
  - A good collection of such methods: https://github.com/strazzere/anti-emulator

Unpacking

The procedure by which the original application code is retrieved from a packed application is called unpacking.

Mainly executed in 2 ways

- Static
  - reproducing, implementing the exact unpacking routine locally
  - Hardest to do but can achieve a 100% correct unpacking
- Dynamic
  - Hooks at the indicated, loading, methods and retrieving the code
  - Memory scrapers
  - Break points on loading methods (easiest, less reliable)

Dynamic unpacking does not guarantee 100% code accuracy or coverage

Each method has advantages and disadvantages

There are cases where an unpack cannot, mathematically, be 100% done to the original code of the application.

Unpacking with jdb

- Start the app in debug mode
  - adb shell am start -D <package_name>/<launcher_activity>
- Get the app PID
  - adb shell ps | grep <package_name>
- Forward the local java debugger PID to the app PID on the device
  - adb forward tcp:<port> jdwp:<app_pid>

JDWP - java debug wire protocol

A more complex analysis of a malware using jdb can be read here

Unpacking with jdb (continued)

- Create a .jdbrc file in the current directory with the content:

```
suspend
stop in java.lang.System.load
stop in dalvik.system.DexFile.loadDex
stop in dalvik.system.DexClassLoader.<init>
stop in dalvik.system.PathClassLoader.<init>
resume
```

- Start the debugger on the designated port
  - jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=<port>
- Retrieve the DEX using adb
- Other useful commands:
  - locals // show local variables
  - where // show stack

Unpacking with Frida

- Push the frida server inside the /data/local/tmp folder on the device (based on the device architecture):
  - adb push frida-server-[x86|x64] /data/local/tmp

- Change server permissions:
  - adb root
  - adb shell chmod 777 /data/local/tmp/frida-server-[x86|x64]

- Run the frida-server:
  - adb shell /data/local/tmp/frida-server-[x64|x86] &

Unpacking with Frida (continued)

- It is based on hooks, that means it intercepts any calls made to the specified Java methods and have the ability to alter their behavior in any way
- Structure of a Javascript Frida hook:

```
const ClassName = Java.use("the_android_class_name");
ClassName.methodName.overload(arguments_types_here).implementation = function() {
    var return_value = this['methodName'].apply(this, arguments);
    return return_value;
}
```

Unpacking with Frida (continued)

- To run the hook you just made you can use the following command:
  - frida –U –f package_name –l hook.js

Now the app should start automatically on the device and the methods will be hooked successfully.