Gleipnir - Android Attack

Exploiting the Android process sharing feature

Sascha Roth

March 30, 2020

1 Abstract

Runtime analysis is a very common technique used by reverse engenieers in order to understand the behaviour of an Application. Developers try to protect their Application by applying runtime protection techniques like: anti-debugging, anti-hooking, root-detection and digest-verification.[1]

According to the android documentation of the

`application` TAG used in the AndroidManifest.xml an App developer has the option to let two apps run inside the same process. In order to achieve that both apps must be signed with the same certificate.

> [...] By setting this attribute to a process name that's shared with another application, you can arrange for components of both applications to run in the same process but only if the two applications also share a user ID and be signed with the same certificate. [...]

[2]
Gleipnir exploit adds a new way how to bypass this rule and neccessary security machanisms of android. This document will give you an idea of how this attack works and a small study to evaluate this finding.

2 Introduction

In general every Android Application can decide wether to enable or disable security features as it likes to be. In Linux every process has it's own memory, calculation time and access permissions. Most of security mechanisms beleive the process they're running belongs to their user (on Android every Application with same signature has it's own user) and is intended for their Application. So they trust in values like

`Context.getFilesDir()` or

`Context.getPackageName()`. In general an Application has to ask itself "Who am I?" and asks the operating system (or trusts in the data the system wrote into their memory on Application startup) to answer this question. Gleipnir exploit tries to tricks the victim Application by taking control over the answer of "Who am I?".

3 Concept

The main part of this exploit is to run the victim Application in a Process of our control in order to access memory, permissions and settings of the target Application. Our own process will be created by install and launching an Application created by us. In this paper this App will be called Gleipnir (see figure 1).

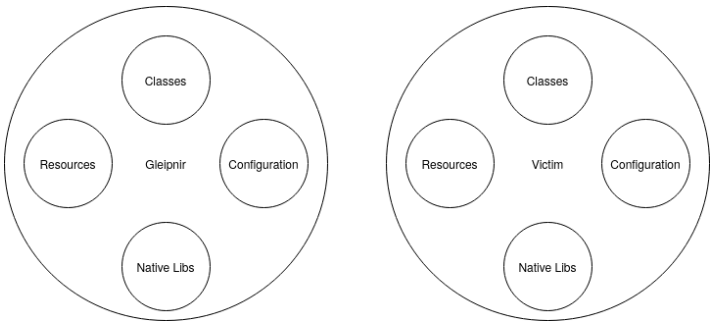

Figure 1: Gleipnir step 1. Gleipnir and the victim are installed.

Once our Application has started we try to load code, resoruces and other dependencies into our process memory (see Figure 2) or our internal storage.

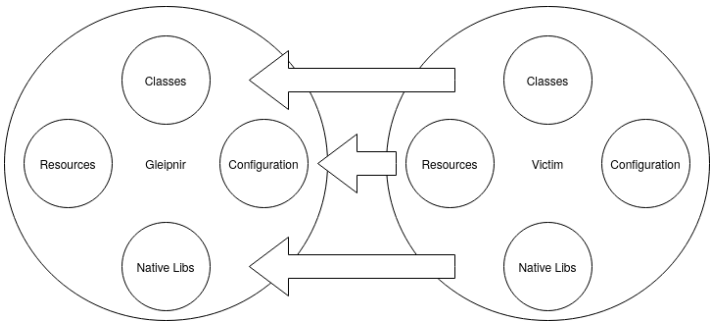

Figure 2: Gleipnir step 2. All required content of victim will be copied or loaded.

At this point the Gleipnir process has Gleipnir's code and resources as well as the victim's content mapped into Gleipnir's scope.

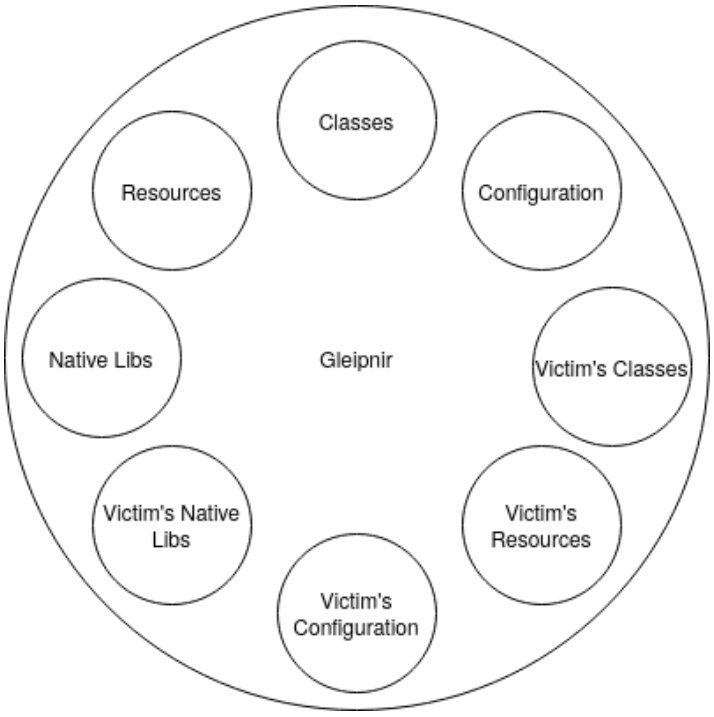

Figure 3: Gleipnir step 3. Gleipnir's and Victim's content is mapped to Gleipnir's scope

The Android Stack of our process has to be manipulated in order to meme to be a process of the victim app. All required paths and names should point to content of the Victim App. Gleipnir won't be able to access it's own resources and components itself anymore, except those which are currently loaded. Ensuring that the code that manipules or traces the viticms application is loaded makes sure that the next step succeeeds.

The next and last step is to launch all components required by the victim Application to start. Here is a list of some components:

* Victim's Application instance
* Service providers
* Services (launched by system intents)
* The Activity with

`android.intent.category.LAUNCHER` category

Now our victim Application should be running within our process and we can debug it, apply native and java hooks without changing the digest of the victim Application.

3.1 Access the victim Application

The

`PackageManger` class provides functions to gather information of other Applications.[12] It is possible to acquire information like: Detailed information about each Activity, permissions, detailed information about each Content provider, detailed information about each service, Application class, data dir, native library directory, process name, apk path and much more meta information.
Gleipnir uses this information to access the victim's apk file, native libraries and meta information required for patching Gleipnir's memory. In general it is not required that the victim Application is installed. Potentially it's possible to gather all required information via the victim APK itself.

3.2 Classes and Classloaders

During Application start the Dalvik executable (DEX)[5] files will be loaded the corresponding ClassLoader. Android uses the

`PathClassLoader` by default which takes the path to the APK file and the native library path as argument (read more in the Native libraries chapter).
Gleipnir replaces the existing ClassLoader instances with new instances using the victim's APK file as parameter. It's very important that the new ClassLoader is empty and doesn't have classes from Gleipnir loaded because support classes or other dependencies can overlap and cause errors while execute the victim app.

3.3 Resources and Assets

Icons, layouts, colors, strings and much more are available via the

`Resources` and

`Assets` class.[7] Every resource has it's unique identifier which is used by the Application while request for it. Also resources have meta information like name and package name which can also be used for identification.[9] Gleipnir replaces all resources and assets occurences with new instances holding the data of the victim app. When components like Activities are started, it's mandatory that the victim resources are loaded peroperly in order to apply the correct theme and layout to be shown.

3.4 Native libraries

Usually native libraries will by loaded by using the

`System.loadLibrary` call.[6] This call will ask the callee's ClassLoader about the path to search for the native library.
So it's required for Gleipnir to set the corresponding native paths while instantiate the new ClassLoaders. When the victim Application gets started it will use these ClassLoaders and all

`System.loadLibrary` calls will search the library at the given place.

3.5 Paths

An Android Application has multiple paths that specify storage, components or other mandatory things. The following table will give an overview about the most important ones and their purpose.
Gleipnir tries to patch those variables to a folder structure it has access permissions.

3.6 Package name

The package name is a unique identifier of an Application. It is one of the most critical values because it has to be used in some cases against the Android operating system where we have to use Gleipnir's package name and in other cases the code of the victim is going access some resources or values where victim's package name is required. Most frequently this is because of the function

`Resources.getIdentifier(..)` which takes a package name as an argument.[9] While retrieve an identifier of an resource of the victim, victim's package name has to be used.

3.7 Components

The following list explaines some important Application components required for starting.

Application The Application instanciated before most other components are started.

Main Activity Will be started as the first Activity while starting the Application from Launcher.

Services Background or foreground services started by broadcast listeners or the Application itself.

Content provider Provides a repository of data for the Application it belongs to or other Applications. [10] Next to the purpose of just sharing data Content providers are also used by third party libraries in order to be initialized by Android itself and not by the Application's code itself. The intention here is to make the integration easier.[11]

Gleipnir starts and initializes every component in order to provide the full Application experience. Lot of libraries or third party components are initialized via Content provider or Application instantiation and will be missing while open the first Activity. Some Applications also using Services during Application usage. In that case the service will be started within the victim's code byself.

3.8 Limitations

As the previous chapters of this attack describe Gleipnir executes the Components of a third party Application inside of Gleipnir's process. The UID in this case will be Gleipnir's uid and will differ from the victim's UID. This causes that the private userdata of the victim Application is unaccessable.[22] For some victim Applications this means a login or backup restore is required.

4 POC

As evaluation of the exploit a POC was created available as zip file "gleipnir-v1.zip". Actually there was lot of forbidden API access and forbidden code execution actions required whose restrictions had to be bypassed. The implementation of this POC is one of multiple ways to achieve the result according to the exploit description. In this chapter we explain two important restriction bypasses used in the POC and also some test against other Applications will be explained.

4.1 Bypass Reflection restrictions

Since API level 28 Android forbidds access to some hidden API functions.[13] Lots of required functions used by the POC are black listed and threw exceptions while trying to access them via the Reflection API.
After investigation on how this restriction works it came out that they check the caller's ClassLoader for system classpath appearence. A very easy and generic solution is to use the Reflection API to call the Reflection API functions for accessing the forbidden functions and fields.

4.2 Bypass Activity AndroidManifest registration

Usually every Activity used within an Android app must be listed in the AndroidManifest.xml file of a project.[14] Technically if an activity is requested to be started, the App requests the ActivityManagerService via

`startActivity`[15] using the Binder [[16], [17]] interface. The ActivityManagerService will check to wich App the requested Activity belongs using the provided AndroidManifest.xml of all Apps. Once the activity was found, the ActivityManagerService forwards a request to the corresponding App to launch the requested Activity.[18] The ActivityThread initializes package context, retrieves some information about the Activity and tells Instrumentation to create a new Activity instance. [4] Figure 4 showcases this procedure with two activities where Activity-A is going to launch Activity-B.

In order to start Activities of our victim Application it is neccessary to bypass this security mechanism. Instead of sending the startActivity() call for the target activity, we send a custom intent for a real existing activity of the Gleipnir application. Once the server processed the request and forwards the event back to our process, we manipulate the incomming event to start the Activity requested by the victim code instead (see more in figure 5). This activity get's fresh new IBinder objects from

`ActivityManagerService` service in order to communicate with other Android services.

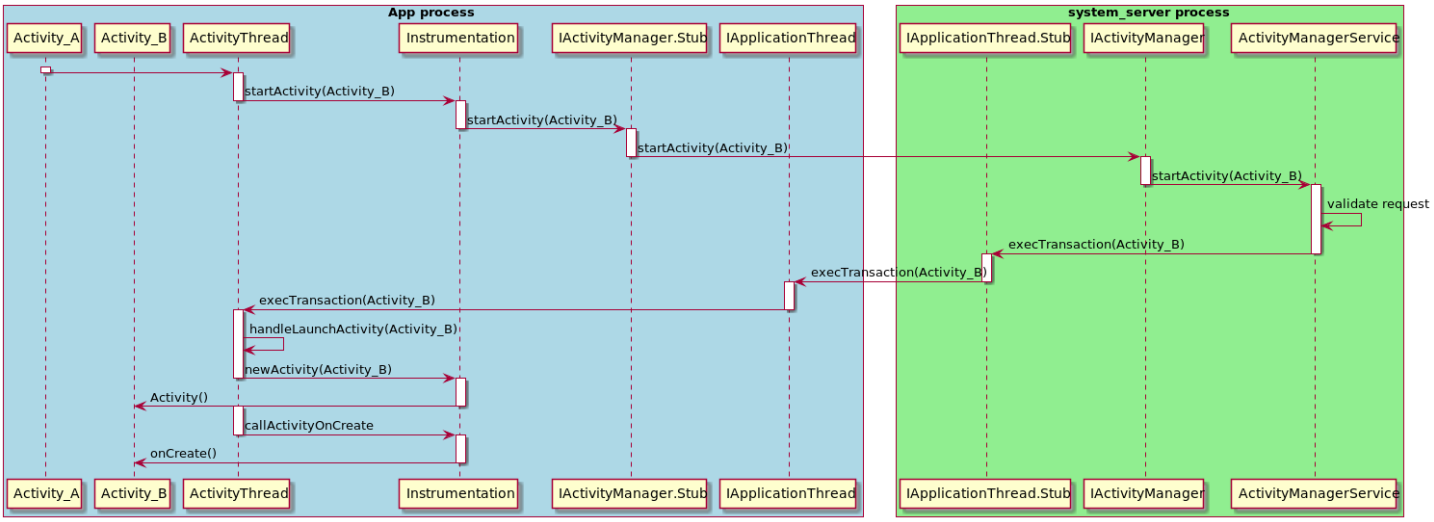

Figure 4: Activity launch normal

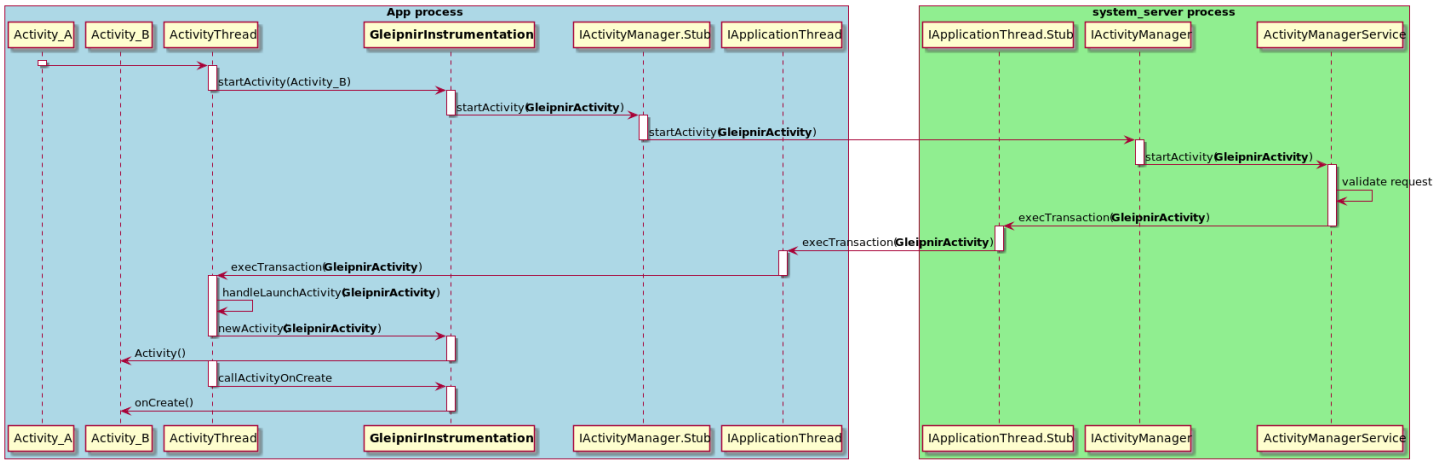

Figure 5: Activity launch bypass

4.3 Study (non-representative)

In order to evaluate the POC and thus the Attack a defined set of Apps downloaded via GooglePlay (https://play.google.com/store/apps) are tested. The total amount of tested Apps counts 99. Figure 6 displays the share of different App categories used for testing.

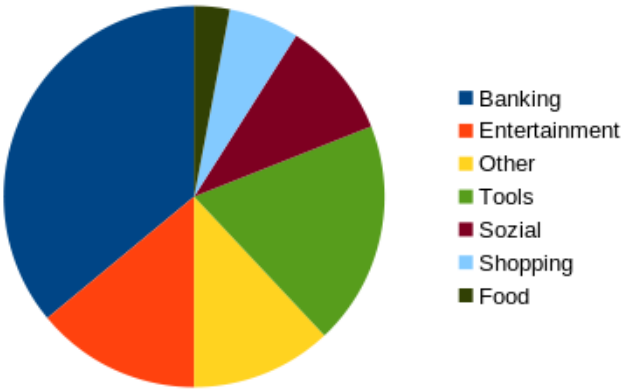

Figure 6: Tested apps overview

Focus of this non representative study are security relevant apps with focus on payment, banking or in-app Buy-ins.[20] Figure 7 is a summary of the test results while using Gleipnir against the given set of Apps.

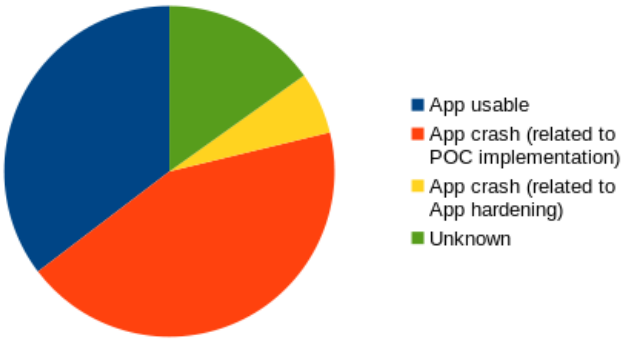

Figure 7: Gleipnir success rate of tested Apps

So 35 percent of apps are directly usable and worked without any complications. Login, writing messages, play pay-tv videos and also receiving push messages was tested successfully.
However 43 percent of apps crashed due to some missing features of the POC. This can be improved with further development and research.
A very less number of Apps (6%) had security features detecting the Gleipnir attack. Most of those features are originally intended to detect repackaging attacks and can by bypassed usually by fixing configuration and paths. Less apps used third party services to verify their integrity. Usually Gleipnir can catch response events of such services and manipulate their content.

5 Conclusion

With the huge number of Android Apps, security has become an important topic and tamper protection is one of the core attacks on Android. This research project was intended to showcase and evaluate a way to launch two differently signed Apps within the same process without using any root permission. In order to achieve that it is required to manipulate the memory or components of the host process and manipulate Binder events sended and received by the host process. The result of the non-representative study didn't give a detailed forecast on how many apps this attack will work out-of-the-box. Indeed it approved that the POC is working and thus the Attack can by applied to various Apps.
Controlled accessed to third party App components like apk or native library files, ClassLoaders checking the signature of a dex file, read only permission to some memory regions would be features to improve the overal situation. Meanwhile, this attack is not the main problem of Android security, this paper is intended to showcase some additional weakpoint to be improved.

References

[1] Michael N. Gagnon, Stephen Taylor, Anup K. Ghosh, *Software Protection through Anti-Debugging*, IEEE, 5th edition, 2007.

[2] Google Inc., *App Manifest Overview*, https://developer.android.com, https://developer.android.com/guide/topics/manifest/application-element, application tag, 2019.

[3] Google Inc. *ActivityManagerService*, https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/am/ActivityManagerService.java, 2020.

[4] Google Inc. *ActivityThread*, https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/ActivityThread.java, 2020.

[5] Google Inc. *Dalvik Executable Format*, https://source.android.com/devices/tech/dalvik/dex-format, 2020.

[6] Google Inc. *System*, https://developer.android.com/reference/java/lang/System, 2020.

[7] Google Inc. *Providing Resources*, https://developer.android.com/guide/topics/resources/providing-resources, 2020.

[8] Google Inc. *Data and storage overview*, https://developer.android.com/training/data-storage, 2020.

[9] Google Inc. *Resources.java*, https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/java/android/content/res/Resources.java, 2020.

[10] Google Inc. *Content provider basics*, https://developer.android.com/guide/topics/providers/content-provider-basics, 2020.

[11] Florent Champign *Your android libraries should not ask for application context*, https://proandroiddev.com/your-android-libraries-should-not-ask-an-application-context-51986cc140d4, 2018.

[12] Google Inc. *PackageManager*, https://developer.android.com/reference/android/content/pm/PackageManager, 2020.

[13] Google Inc. *Restrictions on non-SDK interfaces*, https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces, 2019.

[14] Google Inc. *Activity Element*, https://developer.android.com/guide/topics/manifest/activity-element, 2020.

[15] Google Inc. *IActivityManager*, https://android.googlesource.com/platform/frameworks/base.git/+/refs/heads/master/core/java/android/app/IActivityManager.aidl, 2020.

[16] Google Inc. *Binder*, https://developer.android.com/reference/android/os/Binder, 2020.

[17] Ruhr-Universitt Bochum. *Binder*, https://www.nds.ruhr-uni-bochum.de/media/attachments/files/2012/03/binder.pdf, Thorsten Schreiber, 2012.

[18] Google Inc. *IApplicationThread*, https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/android/app/IApplicationThread.aidl, 2020.

[19] Google Inc. *Instrumentation*,
https://android.googlesource.com/platform/frameworks/base/+/master
/core/java/android/app/Instrumentation.java, 2020.

[20] Google Inc. *Google Play Billing Overview*,
https://developer.android.com/google/play/billing, 2020.

[21] Google Inc. *Application TAG*,
https://developer.android.com/guide/topics/manifest/application-element#proc, 2020.

[22] Avik Chaudhuri *Language-Based Security on Android*,
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.148.2977&rep=rep1&type=pdf, 2009.