GasparZinho

Expanding the native Android Argumentation Framework

Otávio A Araújo Silva

otavioarj

30 Nov. 2024

Agenda

1. Introduction
   - It's time for app R.E?
2. Dynamic execution flow
   - It's all about Frida
3. GasparZinho
   - A LSPosed mod??
   - Not a LSPosed mod?

FAAASTER

Please speed me up to Point 3!!! This presentation will be available on Gaspar' Github repo!!

Introduction

It's time for App R.E?

What's statical reverse engineering of an Android application?

- Unpack the pack The Apk
- Resource/dependencies analysis
- Dex to something? Smali, meta-java? Jadx mostly :)
- Does any dependency is actual a native dependency, shared object?
- Reverse eng shared dependencies? IDA, GHidra?
- ....

Why static reverse engineering? Identify believable execution scenarios, its dependencies, and inference frequent clear secrets (keys, tokens) .

Introduction

Augmentation, argumentation or instrumentation

Simply putting, it's complicate :)

- Augmentation usually relies on critical barriers/superficies exposing: a system that exposes an internal subsystem to another
- Argumentation usually relies on the parametrization of some subsystem, for instance, the use of ptrace for hooking or Java reflection
- Instrumentation, relies on the construction of something external of the analyzed system, like a remote debugger

Hey, what about execution tracing? And execution tainting? ~~Não fode né vei~~[a].

[a] Ooops https://github.com/tiann/KernelSU

Introduction

It's time for App R.E?

What's dynamic reverse engineering of an Android application? Here comes the game:

- Detect its flow of execution, dynamic secrets (generated tokens)
- What's its access to shared resources: sockets, files, memory
- Identify the expected execution flow v.s the available execution flow
- Construct the havoc surface: mostly for attack/abusing

Thus, needing to change, or extract, something from the execution flow.

Dynamic execution flow

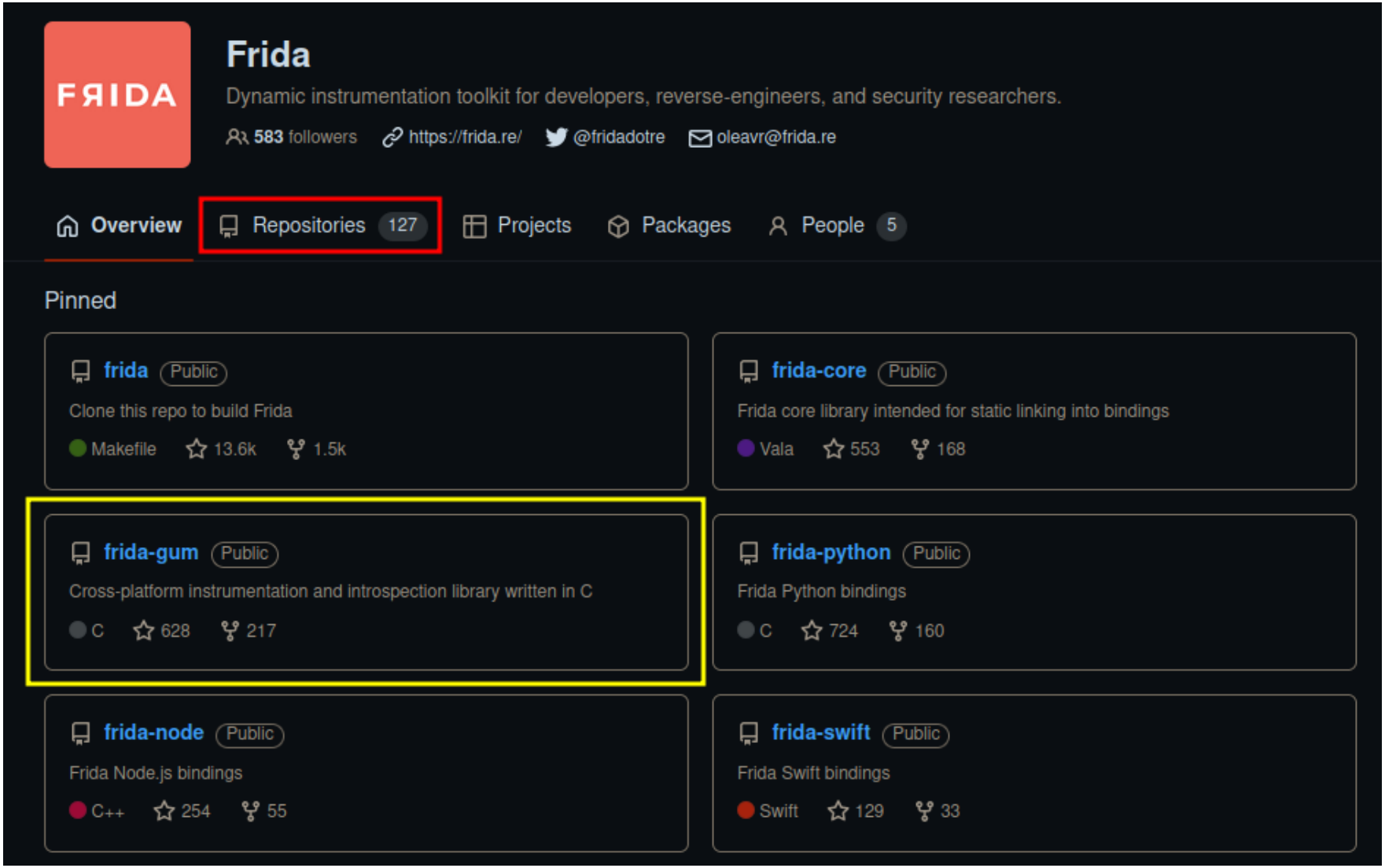
It's all about Frida

Not the first, not the unique, not the faster, not the reliable, but, the most popular.

Maybe it's because Frida runs on many architectures, using a Javascript meta-language, keeping it simple and easy to use?

BUT

- Not the first
- Not the unique
- Not the faster
- **Not the reliable**



Reminder: do not waste time here. JuMp To Gaspar!!

Dynamic execution flow

It's all about GUM

Have you ever compiled Frida? It uses ALOT of dependencies, **127**, and the Gum.

It's basically WHAT does the native hooking, memory scanning, code relocation, and the "stealth" code tracing.

Frida core connects all of the biding, including the Js machine, into the GumJS which is part of Gum.

This "connection" actually is a 4 layer interface, as that:

1. Your Frida code
2. Js machine to meta-Js biding (GumJs)
3. Js biding to core calls
4. Core to Gum operations

Gum

Cross-platform instrumentation and introspection library written in C.

This library is consumed by frida-core through its JavaScript bindings, GumJS.

Provides:

- Instrumentation core
  - Inline hooking: Interceptor
  - Stealthy code tracing: Stalker
  - Memory monitoring: MemoryAccessMonitor
- Cross-platform introspection
  - Running threads and other process state
  - Loaded modules, including their:
    - Imports
    - Exports
    - Symbols
  - Memory scanning
  - DebugSymbol lookups
  - Backtracer implementations
  - Kernel state (iOS only for now)

Dynamic execution flow

It's all about FridaGUM

It works, most of time, but it isn't optimized. Race condition and opportunistic injection is almost out of question.

Anecdotal: rerunning the same Frida script until it works? Having to hook an object creation and failing miserably because the hooking isn't fast enough?

There is no reliable fashion to inject (frida-gadget) into a process without the huge BOOM frida does. Stalker is kind-of stealth, Frida is not. You cannot use one without the other :).

What about ZygiskFrida? It's the same problem, but scalable?

Dynamic execution flow

What about ZygiskFrida?

Pros:

- The gadget is not embedded into the APK itself. So APK Integrity/Signature checks will still pass.
- The process is not being ptraced like it is with frida-server. Avoiding ptrace based detection.
- Control about the injection time of the gadget.
- Allows you to load multiple arbitrary libraries into the process.

Cons: It's the same problem, but scalable, having Frida inside Zygote (through Magisk module) can evade some detection, cool. But it's Frida, stack of Javascript machines to do native code.

Dynamic execution flow

What about alternatives?

Doing exactly what Frida does on Android, with the same power (and beyond?) of interposition in the full Android Runtime? Xposed framework!

Frida is from 2013, first version 1.0.0 went public May 8th in 2013. Xposed was published as a module for Android in March 2012, on XDA Forum :).

Xposed framework?

It's a native ART framework that provides a SDK to be used for argu/agur/instrumentation of Android. It was first created to change Android behavior for custom ROMs.

Dynamic execution flow

Xposed framework

From day one, it was built to be reliable, uptime for multiples applications, changing every aspect of Android. I meant every:

- Taskbar, clock position, shape of battery symbol on taskbar
- How every screen activity swaps, how is the swapping animation
- Sensor filtering, GPS improvement for new algorithm after sensor reading
- Kernel parametrization, CPU governor swap according with user's activities
- ...

It was composed by only extremely optimized ART subsystem, built to don't impact user experience. Later it became from ROM dynamic customization to the number one cheat platform :)

Dynamic execution flow

Xposed frameworkS

Using Java and ARM/AARCH64 as ART, it had many implementation. The most common were:

- Xposed pre Android 5, as kernel driver or system (abandoned)
- Xposed Android 6-8 as part of system (abandoned)
- EdXposed Android 8-11, using Riru Magisk module (abandoned)
- LSPosed_**mod** Android 8.1-15 using Zygisk Magisk module

EdXposed introduced ART hooking framework (initially for Android Pie), leveraging YAHFA or SandHook for native hooking.

Both hooking mechanism were "stealthy" (5 years pre FridaStalker). Not really stealth, but not so loud and messy as Frida hooking :P

Dynamic execution flow

Every release and its implementations used the same SDK as the Xposed (aka OG Xposed), so the coding migration (mainly Java code) was pretty simple.

LSPosed framework

The newest branch of Xposed framework, initially just a port of EdXPosed that was abandoned. It started using YAHFA hooking for all ART operations, and Riru for injection.

Now it uses a more simple, reliable and theoretically hard to detect hooking library, the **Dobby**.

FASTEEEER

Dynamic execution flow

LSPosed framework

By doing inline deoptimization, and intercepting by Java reflection with callback, it can change code without actually tempting on any memory self-integrity; because Java bytecode and DEX processing.

Subsystems

- Zygote mod: The Zygote is the init of Android apps, LSposed uses Zygisk to inject its <1kb module.
- Core: the actual Xposed SDK, which provides all functionality of the framework, except its context placement, in-modules and zygote.
- In-module: a module engine, providing an extensible feature for other system to use the Xposed framework.
- Context: it controls which Android app actually is a mod, which is a target app.

1. Introduction
2. Dynamic execution flow
3. GasparZinho
   - A LSPosed mod??
   - Not a LSPosed mod?

GasparZinho

GasparZinho TURBO?

A LSPosed mod

Uses the LSPosed framework as a module (FOR NOW), relying on its feature to mainly act against an Android app flow, with only low-level argumentation.

Low-level arguwhat?

- Hooking directly and natively the LibC function pre-syscall (Will switch to Dobby directly Dobby-Ga).
- Hooking Java languages (xxtreeemely fast), e.g., in-core file operation and not every file methods for the n-level beyond.
- Per app switch directly on LSPosed context, zero layer from the GasparZinho (FOR NOW).
- Global double chained forbidden/ban/black list, e.g., every hooked function/method that has strings like Frida,posed,gaspar etc., has it removed from the call with the default error of not found.

GasparZinho

```
public class Files extends Gaspar {

    protected static void doFileHooks(Object param ) {
        XC_LoadPackage.LoadPackageParam lpparam = (XC_LoadPackage.LoadPackageParam) param;
        try {
            XposedHelpers.findAndHookConstructor("java.io.File", lpparam.classLoader, "java.lang.String", new XC_MethodHook() {
                @Override
                protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
                    String tmp;
                    for (String filename : denylist) {
                        tmp = (String) param.args[0];
                        if (tmp.contains(filename)) {
//                          printMe(lpparam.packageName, pfile, param.getResult().toString());
                            param.args[0] = UUID.randomUUID().toString();
                            break;
                        }
                    }
                    super.beforeHookedMethod(param);
                }
            });

            XposedHelpers.findAndHookConstructor("java.io.File", lpparam.classLoader, "java.lang.String", "java.lang.String", new XC_MethodHook() {
                @Override
                protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
                    String tmp;
                    for (String filename : denylist) {
                        tmp = (String) param.args[0];
                        if (tmp.contains(filename)) {
//                          printMe(lpparam.packageName, pfile, param.getResult().toString());
                            param.args[0] = UUID.randomUUID().toString();
                            break;
                        }
                    }
                    super.beforeHookedMethod(param);
                }
```

GasparZinho Turbo?

Subsystems?

It was composed in four subsystem, two extensible working as client/server heuristic, in an (undefined?) protocol:

- The Chain: addremove from the list denylist, which is native and ART aware. ~~Partly~~ Fully implemented
- Mods: execute Java and native code provided from an user, like Frida hehe. Not implemented :(- LSPosed design block this.

... and now it will change??!

- In-core: does the low-level argumentation for flawless anti-root/hook/debugger detection. Implemented okm will switch to KernelSU.
- Implementing its own Hidden API (Android API) and uses its own reflection.

In-core native hooks so far?

```
int fake_access(const char *pathname, int mode);
int fake_open(const char* __path, int __flags,  ...);
int fake_openat (int __dir_fd, const char* __path, int __flags, ...);
//FILE* fake_fopen(const char *filename, const char *mode);
void* fake_memmem(const void *haystack, size_t haystacklen,const void *needle, size_t needlelen);
//int fake_memcmp ( const void * ptr1, const void * ptr2, size_t num );
FILE *fake_popen(const char *command, const char *type);
size_t fake__system_property_get(const char *key,char* value);
extern size_t (*back__system_property_get)(const char *key,char* value);
//extern FILE *(*back_fopen)(const char *filename, const char *mode);
extern FILE *(*back_popen)(const char *command, const char *type);
extern void * (*back_memmem)(const void *haystack, size_t haystacklen,const void *needle, size_t needlelen);
//extern int (*back_memcmp) ( const void * ptr1, const void * ptr2, size_t num );
extern int (*back_access)(const char *pathname, int mode);
extern int (*back_open)(const char* __path, int __flags,  ...);
extern int (*back_openat) (int __dir_fd, const char* __path, int __flags, ...);
extern unsigned long (*back_MS_Intune_Instru) (unsigned int param_1,unsigned int param_2,char param_3,char param_4);
unsigned long fake_MS_Intune_Instru (unsigned int param_1,unsigned int param_2,char param_3,char param_4);
#endif //GASPARZINHO_LIBSHOOK_H
```

GasparZinho

In-core native hooks so far!!

```
int fake_access(const char *pathname, int mode);
int fake_faccessat(int dirfd, const char *pathname, int mode, int flags);
int fake_open(const char* __path, int __flags,  ...);
int fake_openat (int __dir_fd, const char* __path, int __flags, ...);
ssize_t fake_getxattr(const char *, const char *, void* , size_t );
int fake_dl_iterate_phdr(int (*)(struct dl_phdr_info *,size_t , void *),void *);
//FILE* fake_fopen(const char *filename, const char *mode);
void* fake_memmem(const void *, size_t ,const void *, size_t );
int fake_memcmp ( const void * ptr1, const void * ptr2, size_t num );
FILE *fake_popen(const char *, const char *);
size_t fake__system_property_get(const char *,char* );
unsigned long fake_MS_Intune_Instru (unsigned int ,unsigned int ,char ,char );
int fake_execve(const char *, char *const  [],char *const  []);
int fake_is_selinux_enabled(void);
extern int (*back_execve)(const char *, char *const  [],char *const []);
extern size_t (*back__system_property_get)(const char *,char* );
//extern FILE *(*back_fopen)(const char *filename, const char *mode);
extern FILE *(*back_popen)(const char *, const char *);
extern void * (*back_memmem)(const void *, size_t ,const void *, size_t );
extern int (*back_memcmp) ( const void * ptr1, const void * ptr2, size_t num );
extern int (*back_access)(const char *pathname, int mode);
extern int (*back_faccessat)(int dirfd, const char *pathname, int mode, int flags);
extern int (*back_open)(const char* __path, int __flags,  ...);
extern int (*back_openat) (int __dir_fd, const char* __path, int __flags, ...);
extern ssize_t (*back_getxattr)(const char *, const char *, void* , size_t );
extern int (*back_dl_iterate_phdr)(int (*)(struct dl_phdr_info *,size_t , void *),void *);
extern unsigned long (*back_MS_Intune_Instru) (unsigned int ,unsigned int ,char ,char );
extern int (*back_is_selinux_enabled)(void);
```

GasparZinho

In-core native hooks so far!! 1/2

```
int fake_access(const char *pathname, int mode);
int fake_faccessat(int dirfd, const char *pathname, int mode, int flags);
int fake_open(const char* __path, int __flags,  ...);
int fake_openat (int __dir_fd, const char* __path, int __flags, ...);
ssize_t fake_getxattr(const char *, const char *, void* , size_t );
int fake_dl_iterate_phdr(int (*)(struct dl_phdr_info *,size_t , void *),void *);
//FILE* fake_fopen(const char *filename, const char *mode);
void* fake_memmem(const void *, size_t ,const void *, size_t );
int fake_memcmp ( const void * ptr1, const void * ptr2, size_t num );
FILE *fake_popen(const char *, const char *);
size_t fake__system_property_get(const char *,char* );
unsigned long fake_MS_Intune_Instru (unsigned int ,unsigned int ,char ,char );
int fake_execve(const char *, char *const  [],char *const  []);
int fake_is_selinux_enabled(void);
extern int (*back_execve)(const char *, char *const  [],char *const []);
extern size_t (*back__system_property_get)(const char *,char* );
//extern FILE *(*back_fopen)(const char *filename, const char *mode);
extern FILE *(*back_popen)(const char *, const char *);
```

In-core native hooks so far!! 2/2

```
extern FILE *(*back_popen)(const char *, const char *);
extern void * (*back_memmem)(const void *, size_t ,const void *, size_t );
extern int (*back_memcmp) ( const void * ptr1, const void * ptr2, size_t num );
extern int (*back_access)(const char *pathname, int mode);
extern int (*back_faccessat)(int dirfd, const char *pathname, int mode, int flags);
extern int (*back_open)(const char* __path, int __flags,  ...);
extern int (*back_openat) (int __dir_fd, const char* __path, int __flags, ...);
extern ssize_t (*back_getxattr)(const char *, const char *, void* , size_t );
extern int (*back_dl_iterate_phdr)(int (*)(struct dl_phdr_info *,size_t , void *),void *);
extern unsigned long (*back_MS_Intune_Instru) (unsigned int ,unsigned int ,char ,char );
extern int (*back_is_selinux_enabled)(void);
```

GasparZinho

Runner Java

```
try {
    runs.exec(Files::doFileHooks);
    //runs.exec(TLS::doMassVerifier);
    runs.exec(Packages::doGetAppInfoUuid);
    runs.exec(Packages::doGetInstalledApps);
    //runs.exec(ExRunner::executeJavaCode);
    runs.exec(Extras::doSELinux);
    runs.exec(Extras::doStringsCmp);
    runs.exec(Extras::doSysProps);
    runs.exec(Extras::doStackTraces);
    runs.exec(Extras::doExec);
    runs.exec(TLS::doTrustManagerImpl);
    runs.exec(TLS::doHttpsURLConnection);
    runs.exec(TLS::doOkHttp);
    runs.exec(()->ClassesL.doClassfind(appcl));
    runs.exec(()->Decomp.decompile(appcl, className: "??....LoginDeeplinkHandler", methodName: "han
    classes=Utils.getClassNames(lpparam);
    for(String cls:classes){
        printMe(cls);
    }
}catch (Throwable e){ throw e;}
```

GasparZinho not a LSPosed mod?

Cool, what's the catch for NOW

Probably it will never be so "easy" to use as Frida.

- It will always demands knowledge of Android ART and native C/C++
- It depends on something else, although LSPosed ~~showed to be reliable, extremely optimized, always evolving~~ zZzZ.
- It depends (NOW) on Magisk, well, Magisk became the "official" Android root module, but it's slowing its release.
- Future is KernelSU solutions (next year?)

Cool, what's the deal

It is extremely fast in comparison with Frida, and waaay more stable:

- LSPosed is a solid software, BUT harder to hide...
- Implementing the use of Dobby internally will fasten up the injection:)
- The more its in-core grows, the less you need to code?

GasparZinho

Here comes the Code&Demo

- Code and more code
- Test with Hidden API detection :)

GasparZinho

Source?

- Clone https://github.com/otavioarj/Gaspar
- Install https://github.com/AGWA/git-crypt
- Import this key for now :(git rebase didn't work.