Android Runtime Restrictions Bypass

Romain Thomas
rthomas@quarkslab.com

Quarsklab — March 22, 2019

Introduction

With the release of Android *Nougat*, Google introduced [1] restriction about native libraries that can be loaded from an Android application. Basically, it limits developers to link against some internal libraries such as `libart.so`.

Later on and with the release of Android Pie, they introduced a new restriction on the access to internal Java methods (or fields). Basically, these restrictions are used to limit developers to access parts of the Android internal framework.

Whereas these limitations aim to be used for compatibility purposes, this article shows how we can take advantage of Android internal to disable them. We briefly explain how these restrictions work and how to disable them from an application without **privileges**.

The first part deals with the native library loading restriction while the second is about Java internal framework restriction.

1 `dlopen()` restrictions

As explained in a blog post [1], Android 7 prevents loading or linking against private libraries such as `/system/lib/libart.so`. It is mainly used to avoid applications relying on libraries and functions that could change in further updates of Android.

Libraries	Target API level	Runtime access via dynamic linker	Android 7.0 (API level 24) behavior	Future Android platform behavior
NDK Public	Any	Accessible	Works as expected	Works as expected
Private (temporarily accessible private libraries)	23 or lower	Temporarily accessible	Works as expected, but you receive a logcat warning.	Runtime error
Private (temporarily accessible private libraries)	24 or higher	Restricted	Runtime error	Runtime error
Private (other)	Any	Restricted	Runtime error	Runtime error

Figure 1 – From `https://developer.android.com`.

This limitation is implemented by associating a namespace for each module[1] loaded in a process. When a module tries to load a library, the Android loader [2] checks if the given module is in the right namespace to load the library. If not, the loader throws an error and does not load the library.

In the case of an Android application, JNI libraries are associated with a namespace that is not allowed to use libraries from `/system/lib`. We can trigger the error by using a JNI function that attempts to open `/system/lib64/libart.so`:

```
void Java_re_android_restrictions_MainActivity_openRestrict(...) {

  void* art_handle = dlopen("/system/lib64/libart.so", RTLD_NOW);

}
```

When this function is executed, we can observe the following error on the logcat output:

```
adb logcat -s "linker:E"
4729  4729 E linker :
  library "/system/lib64/libart.so" ("/system/lib64/libart.so") needed or dlopened by
  /data/app/re.android.restrictions-yALrH==/lib/arm64/librestrictions-bypass.so
  is not accessible for the namespace: [name="classloader-namespace", ld_library_paths="",
  default_library_paths="/data/app/re.android.restrictions/...,
  permitted_paths="/data:/mnt/expand:/data/data/re.android.restrictions"]
```

From this error, we can see that our JNI library is associated with the namespace *classloader-namespace* and is limited to load libraries from the following directories:

- `/data`
- `/mnt/expand`
- `/data/data/re.android.restrictions`

As `libart.so` does not belong to the allowed directories, the library is not loaded and `dlopen` returns a null pointer.

1.1 The `soinfo` structure

All loaded libraries are tied to a structure that contains metadata regarding the context in which the library has been loaded. This structure — named `soinfo` — registers various information like:

- The name of the library (e.g. `libc.so`)
- The path to the library (e.g. `/system/lib64/libc.so`)
- The base address choose by the ASLR (e.g. `0x7f13e09a5000`)
- The **Namespace(s)** associated with the library
- ...

This structure is defined in `linker/linker_soinfo.h` as a part of the Android Bionic module. From this structure, we can notice two fields that are related to the Android namespace implementation:

[1] Library or executable
[2] `/system/bin/linker`

```
struct soinfo {
  ...
  android_namespace_t* primary_namespace_;
  android_namespace_list_t secondary_namespaces_;
  ...
};
```

If somehow, the application manages to access and modify these fields, it can add its **own namespaces** and load libraries from directories that were not allowed.

1.2 soinfo and library mapping

As explained in the previous section, **all** loaded libraries are **tied** to the `soinfo` structure. It turns out that such mapping is performed using an `std::unordered_map` whose the key is a unique id [9] and whose the value is a pointer on the `soinfo` structure.

This mapping is stored in an **exported static** variable named `g_soinfo_handles_map`[3] which is located in **linker** module (`/system/bin/linker`).

From an application point of view, it means that the relative virtual address of this variable can be resolved using the ELF dynamic symbol table. The absolute address can then be computed using `dl_iterate_phdr()` and looking for the base address of `/system/bin/linker`.

Once the application resolved the absolute virtual address of `g_soinfo_handles_map`, it can iterate over the `std::unordered_map` and look for the `soinfo` associated with its JNI library.

```
static const std::string JNI_LIBRARY_NAME = "...";

for (auto&& [hdl, info] : *g_soinfo_handles_map) {
  const char* name = get_soname(info);
  if (name == JNI_LIBRARY_NAME) {
    return info;
  }
}
return nullptr; // Not found
```

Now that the library has a pointer on the `soinfo` structure, it can access to the library namespaces using `get_primary_namespace()`:

```
android_namespace_t* ns = get_primary_namespace(soinfo_ptr);
```

On this pointer, we can call `set_ld_library_paths()` and `set_isolated()` functions to extend the list of allowed directories:

```
ns->set_ld_library_paths({"/system/lib64", "/sytem/lib"});
ns->set_isolated(false);
```

Finally, the loader will not complain about loading a library in `/system/lib64`.

1.3 Summary

To disable the library loading restriction, we took advantage of the fact that the implementation is based on exported static variable that can be access in read / write in the memory space of applications.

[3] `linker/linker_globals.cpp`

Key points

Since Android Nougat, loading libraries can be restricted using *namespace*.

1. All native libraries are associated with a **soinfo** struct.
2. Anyone can **read** and **write** fields of **soinfo**.
3. Native libraries are tied to namespace(s) that is implemented using **soinfo**.
4. We can update this structure with new namespaces.

2 SDK Restriction: hidden API

In Android Pie, Google announced a new restriction to access to the Android internal framework [2]. Basically, this restriction acts as a firewall to grant or deny access to internal **Java methods** or **Java fields**.

Prior to Android Pie, one could use internal framework through JNI or Java reflection:

```
jclass Debug  = env->FindClass("android/os/Debug");
jmethodID mid = env->GetStaticMethodID(Debug, "getVmFeatureList", "()[Ljava/lang/String;");
```

Since Android 9, `getVmFeatureList()` is one of the methods that has been blacklisted and not allowed to accede. Especially, a call to `GetStaticMethodID()` on this method returns a `nullptr` and throw an exception (`NoSuchMethodError`).

2.1 Implementation

This restriction is implemented in the Android runtime (ART) known as *hiddenapi*. By looking at the source code of `GetStaticMethodID()` — located in `jni_internal.cc` — we can see that the function calls `FindMethodID()` which performs checks to allow or deny access to the method:

```
static jmethodID FindMethodID(...) {
  ...
  if (... && ShouldBlockAccessToMember(method, soa.Self())) {
    ...
  }
}
```

`ShouldBlockAccessToMember()` eventually calls `hiddenapi::GetMemberAction()` that ends up with a call to `Runtime::GetHiddenApiEnforcement()`.

Here is an overview of the call stack:

```
                                GetStaticMethodID
                                       ↓
                                  FindMethodID
                                       ↓
                            ShouldBlockAccessToMember
                                       ↓
                           hiddenapi::GetMemberAction
                                       ↓
                            GetActionFromAccessFlags
                                       ↓
                Runtime::Current()->GetHiddenApiEnforcementPolicy()
```

This function can return four values depending on how the restriction is configured:

- **No Checks**: enable access without restriction.
- **Warning**: show warning but allow access.
- **Dark Grey and Black list**: deny access to a dark grey list and a black list.
- **Black list**: deny access to a black list only.

By default, the runtime is configured with the *Dark Grey and Black list* parameter. A library can trigger the restriction by trying to resolve the Java method `android.os.Debug.getVmFeatureList()`:

```
jboolean Java_re_android_hiddenapi_MainActivity_isHiddenApiEnabled(...) {
  jclass Debug  = env->FindClass("android/os/Debug");
  jmethodID mid = env->GetStaticMethodID(Debug, "getVmFeatureList", "()[Ljava/lang/String;");

  if (mid == nullptr) {
    __android_log_write(ANDROID_LOG_VERBOSE, "hiddenapi", "Can't load method!");
    return true;
  }
  return false;
}
```

When this JNI function is executed, we get the following error on the logcat output:

```
W droid.hiddenap: Accessing hidden method
 ↪  Landroid/os/Debug;->getVmFeatureList()[Ljava/lang/String; (dark greylist, JNI)
V hiddenapi: Can't load method!
```

The JNI function is not allowed to access to the internal Java method.

2.2 Turning-off the restriction

In the previous part, we explained that the main check eventually ends-up with a call to `GetHiddenApiEnforcementPolicy()` which is defined in the `art::Runtime` class.

One way to disable the restriction is to force the runtime with a permissive configuration such as `EnforcementPolicy::kNoChecks`.

To change the runtime configuration, the application needs to get an handle on the current instance of the Android Runtime. It can be done by calling the **static** method `art::Runtime::Current()`:

```
#include <runtime/runtime.h>
art::Runtime* current = art::Runtime::Current();
```

However, the downside of this function call is that it requires to **import** the symbol `art::Runtime::instance_`[4] from `libart.so`. Consequently, it makes the JNI library linked against the `libart.so` library. As we explained in the previous part, we will get an error unless we update namespaces.

Another way to get the handle on the Runtime is to use the fact that the first parameter given to `JNI_OnLoad` is an opaque pointer — `JavaVM*` — that is actually a pointer on the internal class `art::JavaVMExt`.

This class is defined in `art/runtime/java_vm_ext.h` and provides an **accessor** on the Runtime:

[4] Actually: `_ZN3art7Runtime9instance_E`

```
namespace art {
class JavaVMExt : public JavaVM {
  ...
  Runtime* GetRuntime() const {
    return runtime_;
  }
}
```

Both functions `art::JavaVMExt::GetRuntime()` and `art::Runtime::Current()` are inlined but the first one returns a class attribute whereas the former uses a static variable.

By using a class attribute, the compiler does not have to import symbols as it just need to compute the offset of the attribute within the class instance:

```
static art::Runtime* my_runtime = nullptr;

jint JNI_OnLoad(JavaVM *vm, void *reserved) {
  my_runtime = reinterpret_cast<art::JavaVMExt*>(vm)->GetRuntime();
  return JNI_VERSION_1_4;
}
```

Once the application got the pointer on the `art::Runtime` object, it can change the current enforcement policy by calling `SetHiddenApiEnforcementPolicy()`:

```
#include <runtime/hidden_api.h> // Expose EnforcementPolicy enum class
...
my_runtime->SetHiddenApiEnforcementPolicy(hiddenapi::EnforcementPolicy::kNoChecks);
```

Moreover, the function `SetHiddenApiEnforcementPolicy()` is **inlined** so this call does not trigger an import and a dependency against `libart.so`.

We now have access to the whole Android framework without restrictions.

Frida Hooking One may want to hook the function `GetHiddenApiEnforcementPolicy()` with Frida to return the wanted value. It turns out that this function is also **inlined**, thus there is not symbols associated with this function. Especially, `Module.findExportByName(...)` doesn't work on this function.

Key points

Starting from Android **Pie**, the runtime is able to restrict access to Android internal framework.

1. Restriction is implemented in `libart.so`.
2. This library is always loaded in the process memory space of applications and we can get a pointer on the `art::Runtime` class.
3. `SetHiddenApiEnforcementPolicy()` can be called to disable the restriction.
4. Inline functions enable to avoid dependencies against `libart.so`.

Recent commits [7, 8] on the ART repository let us suggest that further versions of Android will probably come with an update of the DEX format to enable Android developers to protect their methods and fields using the *hiddenapi* feature.

3 Conclusion

As explained in the documentation about these restrictions, Google does not consider these limitations as **security features**. Nevertheless having a way to disable them is useful while analyzing applications or developing reverse engineering tools for Android.

We sent the details of these techniques to the Android security team [6], they acknowledged quickly and allow us to make them public.

A PoC with sources is available on Quarkslab Github repository:

`https://github.com/quarkslab/android-restriction-bypass`

References

[1] `https://developer.android.com/about/versions/nougat/android-7.0-changes#ndk`

[2] Restrictions on non-SDK interfaces `https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces`

[3] Namespace based Dynamic Linking - Isolating Native Library of Application and System of Android (WANG Zhenhua) `https://tinyurl.com/y8xplzur`

[4] Android Linker Namespace: Security Flaws `https://fadeevab.com/android-linker-namespace-security-flaws/`

[5] VNDK Linker Namespace `https://source.android.com/devices/architecture/vndk/linker-namespace`

[6] `https://issuetracker.google.com/issues/117854171`

[7] `https://android.googlesource.com/platform/art/+/dcfa89bfc06a6c211bbb64fa81313eaf6454ab67/libdexfile/dex/dex_file.h#136`

[8] `https://android.googlesource.com/platform/art/+/dcfa89bfc06a6c211bbb64fa81313eaf6454ab67/libdexfile/dex/dex_file.h#424`

[9] `https://android.googlesource.com/platform/bionic/+/refs/tags/android-9.0.0_r34/linker/linker_soinfo.cpp#753`