Namespace based Dynamic Linking - Isolating Native Library of Application and System of Android

WANG Zhenhua, i@jackwish.net

Abstract

Android provides Software Development Kit (SDK, for Java) and Native Development Kit (NDK, for native language such as C and C++) as public Application Programming Interface (API, includes libraries). For private interface, Java libraries are hidden from applications by Java *ClassLoader*, while native libraries could easily be accessed previously. On the other hand, Project Treble of Oreo, aiming to address fragmented ecosystem by dividing Android implementation into Framework and Vendor part, needs to manage native libraries of two sets in one process separately. With these challenges, Android dynamic linker introduces *namespace* (1, 2 and 3) which isolates dynamic linking space. Android system deploys *namespace* to prevent applications from dynamically linking against private native libraries, and host Framework and Vendor libraries in different sandboxes. This article analyzes the *namespace* of Android Oreo, including the mechanism of dynamic linker as well as its inter-cooperation with high level *namespace* policy, and discusses the impacts and benefits.

Introduction

Problem of Private Native Library

Android, dominating the most mobile devices, releases SDK and NDK with which a great variety of applications have been developed. Some applications (see section *Misbehaving Applications*) are not satisfied with public API such that they play with private libraries which are system's libraries other than SDK and NDK.

For Java, *ClassLoader* guarantees a type-safety execution environment for applications. The hierarchy of *ClassLoader* hides non-SDK resource from applications' access. On the contrary, in native world, Android lacks ready-made mechanism to restrict private library usage since Linux, on which Android is based, focuses on user/kernel data protection. While Android is open sourced, it's prone for developers to make use of "powerful" private library.

Nevertheless, the interfaces of private libraries are unstable as they may change across Android release without any public notification. Applications that depending on legacy private interfaces may fail on newly-released Android, which is not welcomed. Therefore, forbidding Android applications' access to private native library is critical to ecosystem.

Problem of Project Treble

In addition, Android platform is highly fragmented regardless of its popularity. Device-makers are not willing to upgrade legacy devices to new Android platform as it usually requires significant efforts; meanwhile, developers are pushed to test their applications on massive devices as there are so many legacy platforms among them. Fragmented platform discourages everyone in this ecosystem.

In Android Oreo, Google announces Project Treble to address the fragment issue. Treble divides Android platform to be Framework and Vendor part, which hosts applications and manages device-specific features respectively. When handling device-specific functionality, Framework asks Vendor for service via Vendor interface, an interface that expected to be stable across several Android releases. Thus with Treble, Android Framework can upgrade while the Vendor implementation remains unchanged, as Figure 1. In this way, vendors can easily upgrade legacy devices and applications can benefit from latest Framework features.

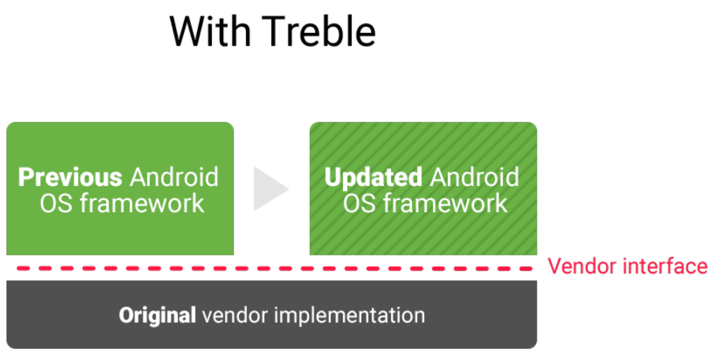
With Treble
Previous Android OS framework
Updated Android OS framework
Vendor interface
Original vendor implementation

Figure 1: Project Treble: Updating Framework without Change to Vendor Implementation.

For the perspective of native library, Treble introduces two sets of native libraries in one process - the Framework's and Vendor's. In some scenario, libraries in these two sets may have same name but different implementation. Since library symbols are exposed to all code of one process traditionally, these two sets need to be distinguished and linked separately.

The Namespace Proposal

To mitigate these problems, Android dynamic linker introduces *namespace* based dynamic linking, which isolates library loading space in native - similar with the resource separation of *ClassLoader* in Java. With this design, each library is loaded into one specific namespace where cannot access libraries in other namespaces unless they are shared through a *namespace link*.

This article reveals the mechanism of namespace based dynamic linking first. After that, the namespace policy, which hides private library from applications and manages Framework's and Vendor's own libraries, of Android system is illustrated. We discuss impact and benefit also.

The Mechanism of Namespace based Dynamic Linking

Namespace is the most significant change of dynamic linker in Android Nougat and Oreo. This section summarizes namespace based dynamic linking, which loads native libraries into sandboxes and shares libraries through *namespace link*.

Dynamic Linking Basic

Dynamic linker locates shared libraries from storage, loads them into memory and links their symbols at runtime.

Whenever received library loading request, dynamic linker traverses *library search path* (LSPath) to search that library. The LSPath on Solaris and Linux is configured by variable `LD_LIBRARY_PATH` which is supposed to contain directories where shared libraries are stored. Dynamic linker parses `LD_LIBRARY_PATH` to be LSPath at the beginning of a process, and LSPath is unlikely to change during the process's lifetime.

Loaded libraries are tracked by dynamic linker in *already-loaded library list* (ALList) for further usage such as symbol lookup. If the library requested to be loaded has already been in ALList, the existing data will be used directly - speeding up dynamic linking. As process is running, ALList represents the status of native code resource somehow.

In one process, all native code has equal access to shared libraries under LSPath; meanwhile, all loaded libraries are registered in one same ALList, as the left part of Figure 2. (Refer to *Android Dynamic Linker in Marshmallow* for further detail).

Isolated Namespaces

Starting with Nougat, Android dynamic linker divides library loading space of one process into several namespaces. Each namespace has its own LSPath and ALList.

At runtime, dynamic linker creates namespaces per request and assigns LSPath which could be varied among namespaces. Library loading in one namespace will only search LSPath of that namespace. Take Figure 2 as example, consider that there are two namespaces `namespace 1` and `namespace 2` of which the LSPath are `path1` and `path2` respectively. As dynamic linker searches library in LSPath of specific namespace, `namespace 1` can only load library under `path1` while `namespace 2` can only load library under `path2`.

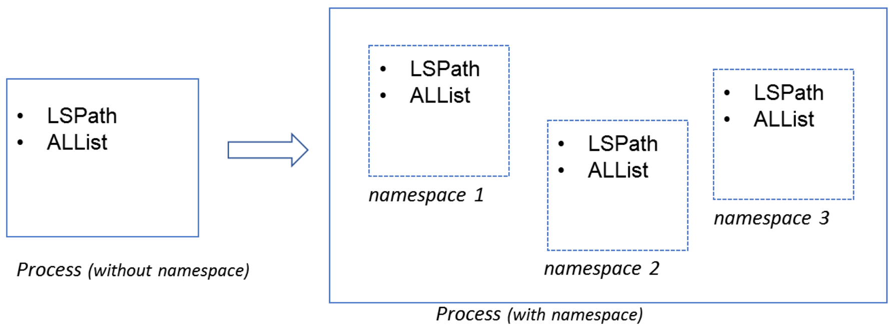

Figure 2: Dynamic Linking Scenario: without and with Namespace

Android dynamic linker provides namespace API with which user can define their own namespace policy to isolate native libraries that are located in different directories. Among these API, `android_create_namespace()` creates namespace with specific LSPath, and `android_dlopen_ext()` loads library in specific namespace. In addition, in `android_create_namespace()`, user may declare whether a namespace is "strictly isolated" (flag `ANDROID_NAMESPACE_TYPE_ISOLATED`) and "permitted path". Non-strictly isolated namespace accepts any absolute path while strictly isolated one only accept absolute path points to library under LSPath or permitted path. We will focus on LSPath in following discussion.

For library that uses conventional dynamic linking API, e.g. `dlopen()`, dynamic linker loads new library in the caller's namespace. While dynamic linker manages a `(default)` namespace by default, a process without namespace usage lives in this namespace. Dynamic linking in such scenario is same as conventional. Thereby, namespace is backward-compatible and transparent to these shared libraries.

With this design, libraries are loaded in isolated namespaces where libraries in other namespaces are invisible to them. Nevertheless, considering the reality of operating system, there are some libraries that need to be visible to all code of one process naturally, e.g. NDK. In Android Oreo, *namespace link* is introduced to share libraries across namespaces.

Namespace Link

Namespace link is one-way *link* that created between two namespaces to share libraries from one to another. In Figure 3, the yellow library nodes are shared from a namespace while gray nodes in that namesapce are not shared. These libraries are shared by file name (expected to be same as `SONAME`) among namespaces. A namespace may have multiple *namespace links*, as the most left node in Figure 3; and a namespace can be linked from multiple namespaces, as the central node. A namespace may share different sets of libraries via the *namespace links* that linked to it. Linked namespace can link to other namespaces, node 1 and the central node in Figure 3 are such examples.

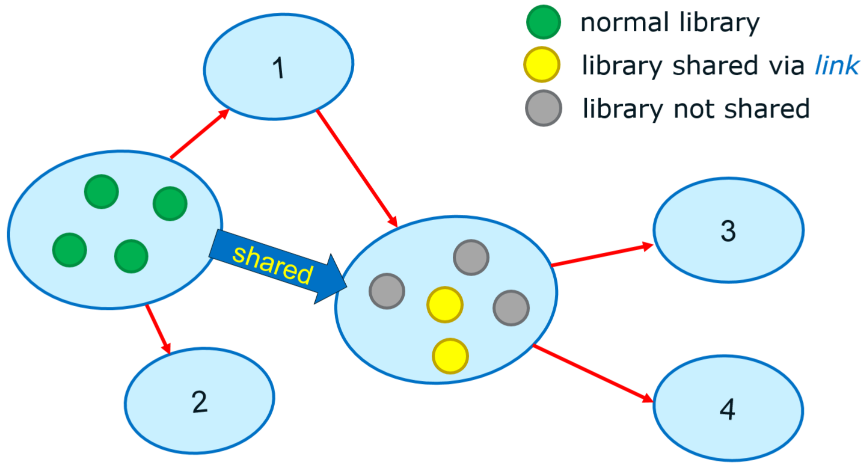

Figure 3: Namespace link: share libraries from one namespace to another

When loading library, the namespace in use is firstly searched; if fail, dynamic linker walks *namespace links* that share the library, and tries to load in the linked namespaces. Library and its dependency that loaded in linked namespace won't be added to ALList of the namespace that it shared to. The dependency remains invisible to other namespaces unless some ones are shared through other *namespace links*, as Figure 3.

By deploying *namespace link*, libraries are shared across isolated namespaces elegantly. Besides the generic design, dynamic linker creates two namespaces for special purpose.

Special Namespaces

When Android dynamic linker is initializing, it bootstraps namespace by creating the first namespace - `(default)` namespace. `(default)` namespace is not "strictly isolated" and LSPath of it is parsed from `LD_LIBRARY_PATH`. Like stated in section *Isolated Namespaces*, if nobody leverages namespace API in one process, `(default)` namespace will be the only one namespace where all libraries are loaded. Namespace degenerates into conventional dynamic linking in such scenario.

Android dynamic linker creates `(anonymous)` namespace in API `android_init_anonymous_namespace()`. Although each library belongs to a namespace, JIT (Just In Time) code, e. g. Mono, doesn't belong to any library or namespace. For `dlopen()` in JIT-like code, new library will be loaded in `(anonymous)` namespace. So, `(anonymous)` namespace is designed to hold JIT-like code and consequently loaded library. Namespace user is expected to call `android_init_anonymous_namespace()` actively; `(default)` namespace is taken as `(anonymous)` otherwise.

The Namespace Policy of Android System

To fulfill the target of forbidding application's access to private library and managing Framework and Vendor's library separately, Android system deploys the namespace mechanism in different way.

NativeLoader: Mapping ClassLoader to Namespace

NativeLoader is the Android component that forbids library access and shares library across namespaces by utilizing namespace mechanism.

Android applications load native library in two ways: `System.loadLibrary()` in Java and `dlopen()` or `android_dlopen_ext()` in native. Regarding Java, `System.loadLibrary()` eventually calls into NativeLoader which will load the library in the namespace that mapped by the ClassLoader instance of the `System.loadLibrary()` context. On the other hand, library loading in native calls Android dynamic linker (`libdl.so` in NDK provides the dummy API) directly.

NativeLoader maintains the mapping from ClassLoader to native namespace. Figure 4 is a simplified hierarchy of Java ClassLoader (left half) and namespace (right half) that NativeLoader manages. A namespace is created at the first

`System.loadLibrary()` inside classes loaded by one same ClassLoader. One exception is the first namespace actively created by ApplicationLoader (`CL NS 0` mapped by `Path CL` in Figure 4 for example) when an application is starting. NativeLoader names all these namespaces as `classloader-namespace` and links them to `(default)` namespace, applications have access to same set of system libraries therefore. The libraries that shared via these *namespace links* are parsed from `/etc/public.libraries.txt` and `/vendor/etc/public.libraries.txt` which SoC/device vendors may extend. Since native namespaces are one-to-one mapped by ClassLoaders, they form the same hierarchy. In Figure 4, `App CL 1` maps to `CL NS 1` and `App CL 2` maps to `CL NS 2`. Nevertheless, some ClassLoader doesn't have peered namespace because it uses no native library, e.g. node `App CL 3`.

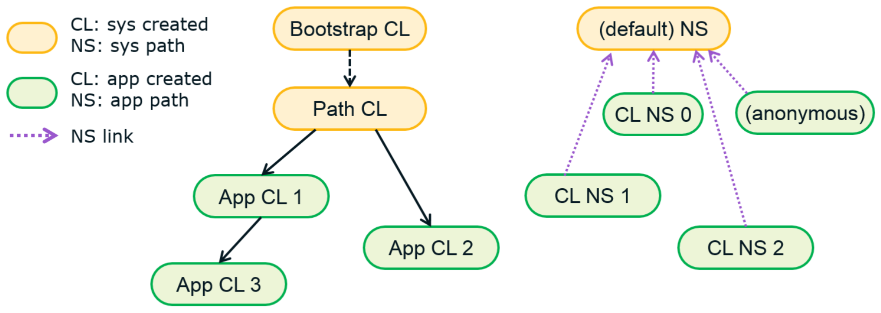

Figure 4: Namespace Hierarchy Managed by NativeLoader

In this way, applications' access to private system libraries, the ones that are other than list `/etc/public.libraries.txt`, are forbidden.

Treble: Managing Framework and Vendor Library

Treble divides Android to be Framework part and Vendor part which are maintained separately by Framework vendor (e.g. Google and Xiaomi) and SoC/device Vendor (e.g. Qualcomm and Samsung). The Vendor Interface in Figure 1 includes Hal Interface Definition Language (HIDL, interpreted as Hardware Interface Definition Language sometimes), Vendor NDK (VNDK) and so on. HIDL is a scheme defining how Vendor serves Framework.

VNDK is a set of Framework libraries that Vendor can depend on when implementing their functionality. In Treble, if Framework updates while Vendor does not, Android system retains legacy VNDK libraries in a separate directory, `/system/lib/nvdk-sp` for example, such that Vendor still works with them flawlessly. If Framework and Vendor are the same version, the VNDK used by them are the same too; otherwise, they use different VNDK.

Android system manages these scenarios by creating `sphal` and `vndk` namespaces, where all Vendor libraries are loaded (by `libvndksupport.so` or similar logic) and legacy VNDK libraries are loaded. `sphal` namespace is linked to `vndk` namespace in case legacy VNDK implementation is required.

The Ecosystem and Solutions

Namespace impacts many applications which depend on non-NDK library. We take a glance at these applications and look into the temporary backward compatibility solutions.

Misbehaving Applications

Instagram (in `com.instagram.android` of version `10.3.2`, `libigbitmap_runtime_for_v21.so` and `libigbitmap_runtime_for_v23.so` depend on `libandroid_runtime.so`) is an example of depending private library `libandroid_runtime.so`. According to namespace mechanism and policy of Android, the library loading shall fail, and application will crash if it doesn't ignore `UnsatisfiedLinkError` - which is the most case. However, there are too many applications linking against private libraries, thus forbidding them all is too aggressive. Alternatively, *greylist* (will address in section *The Greylist Library*) tolerates some legacy misbehaving applications temporarily.

Messenger (in `com.facebook.orca` of version `95.0.0.20.70`, `libcpp_helper.so` tries to `dlopen(libart.so)`) is an example of loading private library `libart.so`. As `libart.so` is not part of *greylist*, meaning that Messenger shall fail. However, this behavior is so popular among Facebook's applications and Facebook is such a powerful application vendor that it seems Google failed to reach an agreement with Facebook on `libart.so` issue. Eventually, Android adds `/system/fake-libs` as part of LSPath when creating all `classloader-namespace` namespace; and creates a fake `libart.so` which only prints some log to work around Facebook applications.

The *Greylist* Library

Android system engineers produced the most popular private libraries (*greylist* libraries) used among misbehaving applications on Play Store to minimize namespace's impact to ecosystem. When loading library in one namespace, Android dynamic linker will try LSPath of `(default)` namespace if fail to load a *greylist* library in current namespace and linked namespaces.

Greylist only works for applications targeting earlier Android system (before Nougat). A *greylist* library will be loaded if the legacy application loads or depends on it but doesn't pack one by itself. The dependency of a *greylist* will all become *greylisted* (except the NDK ones) thus the library loading will succeed. *Greylist* libraries are loaded in current namespace directly - a copy in memory that different from the ones in `(default)` namespace.

In some scenarios, access to non-*greylist* private library may also succeed. For example, `dlopen(libandroidfw.so)` after `dlopen(libandroid_runtime.so)` will pass since `libandroidfw.so`, which is a non-*greylist* library in the dependency of *greylist* library `libandroid_runtime.so`, has already been loaded in this namespace when loading `libandroid_runtime.so`. This behavior should be out the intention of *greylist*. Anyway, *greylist* is a temporary backward compatibility solution that will be removed in future Android release.

The Memory Usage

Lastly, we address the additional memory consumption introduced by namespace.

Classic dynamic linking, as referred already, library loading is conducted in the scope of process. For any library in storage, there is no more than one copy in memory. Namespace based dynamic linking, however, might load different library copy into memory according to application behavior.

The additional memory consumption is because most library resource are not shared across namespaces. Typical example is *greylist* library. Consider an application targeting Android Marshmallow has three `classloader-namespace` namespaces, say `ns1`, `ns2` and `ns3`, where library `lib1.so`, `lib2.so` and `lib3.so` are loaded respectively. These three libraries are all dependent on `libandroid_runtime.so` which is a *greylist* library. Therefore, there is one copy of `libandroid_runtime.so` and its dependency in memory for each of these three namespaces in addition to the one in `(default)` namespace.

For applications which create multiple namespaces and load *greylist* library, a great deal of additional memory will be consumed.

Summary

Carrying the mission to enforce Android applications to utilize public system library only and mange Framework and Vendor library apart in Project Treble, Android introduces namespace based dynamic linking and proper usage policy.

Android dynamic linker isolates dynamic linking scope of one process into namespaces. The library loading in each namespace is same as conventional dynamic linking in one process. Thereby, libraries in one namespace are hidden from other namespace. Meanwhile, *namespace link*, which links one namespace to another, share specific libraries across namespaces.

By maintaining namespace according to Java ClassLoader and share public libraries to these namespaces, Android system isolates native libraries of system's and application's. In addition, *greylist* which grants application's access to some particular private libraries, is introduced to work-around misbehaving applications temporarily. As a side effect, *greylist* increases memory consumption of native library.

By creating `sphal` namespace and `vndk` namespace where Vendor library and VNDK library are loaded, Android system can manage Framework and Vendor library separately in one process even if they are not of same version in Treble. The native library layering and the interaction between Framework and Vendor is complex.

All in all, namespace is an effective scheme which isolates and shares native resource efficiently and elegantly. It could be deployed into other modern operating systems to block abusing applications, thus to improve the ecosystem both for system developers and application vendors.