前言

在过完了四大组件的容器内实现后，最后我们打算着重看看之前只是模糊提了几句的重定向问题，看看容器是如何把相关的资源和路径进行重定向的。

路径重定向和 Xposed 注入时机分析

在之前分析 Activity 时有这么一行代码用来支持路径的重定向：

```
IOCore.get().enableRedirect(packageContext);
```

代码注释中可以看到，应用本身的资源其实已经通过 Application 完成重定向了，但是仍有一些通过硬编码路径来访问文件的情况，为了避免这类访问引发崩溃，因此需要把它们重定向到新路径：

```
// 由于正常情况Application已完成重定向，以下重定向是怕代码写死。
public void enableRedirect(Context context) {
  Map<String, String> rule = new LinkedHashMap<>();
  String packageName = context.getPackageName();

  try {
    ApplicationInfo packageInfo =
        SandBoxCore.getBPackageManager()
            .getApplicationInfo(
                packageName, PackageManager.GET_META_DATA,
BActivityThread.getUserId());
    int systemUserId = SandBoxCore.getHostUserId();
    rule.put(String.format("/data/data/%s/lib", packageName),
packageInfo.nativeLibraryDir);
    rule.put(
        String.format("/data/user/%d/%s/lib", systemUserId, packageName),
        packageInfo.nativeLibraryDir);

    rule.put(String.format("/data/data/%s", packageName),
packageInfo.dataDir);
    rule.put(String.format("/data/user/%d/%s", systemUserId, packageName),
packageInfo.dataDir);

    if (SandBoxCore.getContext().getExternalCacheDir() != null
        && context.getExternalCacheDir() != null) {
      File external =
BEnvironment.getExternalUserDir(BActivityThread.getUserId());
```

```
      // sdcard
      File sdcardAndroidFile = new File("/sdcard/Android");
      String androidDir = String.format("/storage/emulated/%d/Android",
systemUserId);
      if (!sdcardAndroidFile.exists()) {
        sdcardAndroidFile = new File(androidDir);
      }
      if (sdcardAndroidFile.exists()) {
        File[] childDirs = sdcardAndroidFile.listFiles(pathname ->
pathname.isDirectory());
        if (childDirs != null) {
          for (File childDir : childDirs) {
            Log.d(TAG, childDir.getAbsolutePath());
            rule.put(
                "/sdcard/Android/" + childDir.getName(),
                external.getAbsolutePath() + "/Android/" +
childDir.getName());
            rule.put(
                androidDir + "/" + childDir.getName(),
                external.getAbsolutePath() + "/Android/" +
childDir.getName());
          }
        } else {
          rule.put("/sdcard/Android", external.getAbsolutePath() +
"/Android");
          rule.put(androidDir, external.getAbsolutePath() + "/Android");
        }
      } else {
        rule.put("/sdcard/Android", external.getAbsolutePath());
        rule.put(androidDir, external.getAbsolutePath());
      }
    }
    if (SandBoxCore.get().isHideRoot()) {
      hideRoot(rule);
    }
    proc(rule);
  } catch (Exception e) {
    e.printStackTrace();
  }
  for (String key : rule.keySet()) {
    get().addRedirect(key, rule.get(key));
  }
  NativeCore.enableIO();
}
```

总的来说，先初始化了一些重定向规则，包括：

```
rule.put(String.format("/data/data/%s/lib", packageName),
packageInfo.nativeLibraryDir);
rule.put(
    String.format("/data/user/%d/%s/lib", systemUserId, packageName),
    packageInfo.nativeLibraryDir);
rule.put(String.format("/data/data/%s", packageName), packageInfo.dataDir);
rule.put(String.format("/data/user/%d/%s", systemUserId, packageName),
packageInfo.dataDir);
rule.put(
    "/sdcard/Android/" + childDir.getName(),
    external.getAbsolutePath() + "/Android/" + childDir.getName());
rule.put(
    androidDir + "/" + childDir.getName(),
    external.getAbsolutePath() + "/Android/" + childDir.getName());
```

除此之位还有对进程路径的重定向：

```
private void proc(Map<String, String> rule) {
  int appPid = BActivityThread.getAppPid();
  int pid = Process.myPid();
  String selfProc = "/proc/self/";
  String proc = "/proc/" + pid + "/";

  String cmdline = new File(BEnvironment.getProcDir(appPid),
"cmdline").getAbsolutePath();
  rule.put(proc + "cmdline", cmdline);
  rule.put(selfProc + "cmdline", cmdline);
}
```

相当于把对这些路径的访问全部过滤成后面的那个参数，然后再把它们注入到 Native 层去：

```
public void addRedirect(String origPath, String redirectPath) {
  if (TextUtils.isEmpty(origPath)
      || TextUtils.isEmpty(redirectPath)
      || mRedirectMap.get(origPath) != null) return;
  // Add the key to TrieTree
  mTrieTree.add(origPath);
  mRedirectMap.put(origPath, redirectPath);
  File redirectFile = new File(redirectPath);
  if (!redirectFile.exists()) {
    FileUtils.mkdirs(redirectPath);
  }
```

```
    NativeCore.addIORule(origPath, redirectPath);
  }
```

最后激活一下这些规则：

```
void enableIO(JNIEnv *env, jclass clazz) {
    IO::init(env);
    nativeHook(env);
}

void nativeHook(JNIEnv *env) {
    BaseHook::init(env);
    UnixFileSystemHook::init(env);
    VMClassLoaderHook::init(env);
//    RuntimeHook::init(env);
    BinderHook::init(env);
}
```

可以看到，容器对三个对象做了重定向，分别是文件系统、ClassLoader 以及 Binder。

文件系统 Hook

我们从文件系统看起：

```
void UnixFileSystemHook::init(JNIEnv *env) {
  const char *className = "java/io/UnixFileSystem";
  JniHook::HookJniFun(env,
                      className,
                      "canonicalize0",
                      "(Ljava/lang/String;)Ljava/lang/String;",
                      (void *) new_canonicalize0,
                      (void **) (&orig_canonicalize0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "getBooleanAttributes0",
                      "(Ljava/lang/String;)I",
                      (void *) new_getBooleanAttributes0,
                      (void **) (&orig_getBooleanAttributes0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "getLastModifiedTime0",
                      "(Ljava/io/File;)J",
                      (void *) new_getLastModifiedTime0,
                      (void **) (&orig_getLastModifiedTime0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "setPermission0",
                      "(Ljava/io/File;IZZ)Z",
                      (void *) new_setPermission0,
                      (void **) (&orig_setPermission0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "createFileExclusively0",
                      "(Ljava/lang/String;)Z",
                      (void *) new_createFileExclusively0,
                      (void **) (&orig_createFileExclusively0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "list0",
                      "(Ljava/io/File;)[Ljava/lang/String;",
                      (void *) new_list0,
                      (void **) (&orig_list0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "createDirectory0",
                      "(Ljava/io/File;)Z",
                      (void *) new_createDirectory0,
                      (void **) (&orig_createDirectory0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "setLastModifiedTime0",
                      "(Ljava/io/File;J)Z",
                      (void *) new_setLastModifiedTime0,
                      (void **) (&orig_setLastModifiedTime0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "setReadOnly0",
                      "(Ljava/io/File;)Z",
                      (void *) new_setReadOnly0,
                      (void **) (&orig_setReadOnly0),
                      false);
  JniHook::HookJniFun(env,
                      className,
                      "getSpace0",
                      "(Ljava/io/File;I)J",
                      (void *) new_getSpace0,
                      (void **) (&orig_getSpace0),
                      false);
}
```

可以看到主要就是对这些函数做了 Hook，以及 Hook 的方式也很朴素：

```
void
JniHook::HookJniFun(JNIEnv *env,
                    const char *class_name,
                    const char *method_name,
                    const char *sign,
                    void *new_fun,
                    void **orig_fun,
                    bool is_static) {
  if (HookEnv.art_method_native_offset == 0) {
    return;
  }
  jclass clazz = env->FindClass(class_name);
  if (!clazz) {
    ALOGD("findClass fail: %s %s", class_name, method_name);
    env->ExceptionClear();
    return;
  }
  jmethodID method = nullptr;
  if (is_static) {
    method = env->GetStaticMethodID(clazz, method_name, sign);
  } else {
    method = env->GetMethodID(clazz, method_name, sign);
  }
  if (!method) {
    env->ExceptionClear();
    ALOGD("get method id fail: %s %s", class_name, method_name);
    return;
  }
  JNINativeMethod gMethods[] = {
      {method_name, sign, (void *) new_fun},
  };

  auto artMethod =
      reinterpret_cast<uintptr_t *>(GetArtMethod(env, clazz, method));
  if (!CheckFlags(artMethod)) {
    ALOGE("check flags error. class: %s, method: %s", class_name, method_name);
    return;
  }
  *orig_fun =
      reinterpret_cast<void *>(artMethod[HookEnv.art_method_native_offset]);
  if (env->RegisterNatives(clazz, gMethods, 1) < 0) {
    ALOGE("jni hook error. class: %s, method: %s", class_name, method_name);
    return;
  }
  // FastNative
  if (HookEnv.api_level == __ANDROID_API_O__
      || HookEnv.api_level == __ANDROID_API_O_MR1__) {
    AddAccessFlag((char *) artMethod, kAccFastNative);
  }
  ALOGD("register class: %s, method: %s success!", class_name, method_name);
}
```

先用 GetStaticMethodID 或 GetMethodID 来获取原始函数的位置，然后把它回填到 orig_fun 中，最后调用 RegisterNatives 用新函数注册进去，替换结束。

而新函数基本上都是这样实现的：

```
HOOK_JNI(jboolean, getSpace0, JNIEnv *env, jobject obj, jobject file, jint t)
{
  jobject redirect = IO::redirectPath(env, file);
  return orig_getSpace0(env, obj, redirect, t);
}
```

先过滤一遍路径，然后调用原函数。过滤流程最终会调用这个函数：

```
const char *IO::redirectPath(const char *__path) {
  list<IO::RelocateInfo>::iterator iterator;
  for (iterator = relocate_rule.begin(); iterator != relocate_rule.end();
++iterator) {
    IO::RelocateInfo info = *iterator;
    if (strstr(__path, info.targetPath) && !strstr(__path, "/sandbox/")) {
      char *ret = replace(__path, info.targetPath, info.relocatePath);
      ALOGD("redirectPath %s  => %s", __path, ret);
      return ret;
    }
  }
  return __path;
}
```

就是遍历一下找到合适的规则然后根据规则做替换，没啥别的内容了。不过这里有一个疑问，看起来程序并没有对 open、stat 之类的 libc 函数做 Hook，那如果后续有硬编码写死去调用 open 的情况是不是就会出错呢？

同样的，除此之外还有很多各种各样的函数被调用时参数中都会附带路径，这些函数不需要一起过滤吗？以及同时过滤这么多函数也很容易被检查发现也是一个问题。或许后续需要完善。

ClassLoader Hook

```
void VMClassLoaderHook::init(JNIEnv *env) {
    const char *className = "java/lang/VMClassLoader";
    JniHook::HookJniFun(env, className, "findLoadedClass", "
(Ljava/lang/ClassLoader;Ljava/lang/String;)Ljava/lang/Class;",
                        (void *) new_findLoadedClass,
                        (void **) (&orig_findLoadedClass), true);
}
```

挺朴素的，跟前文的方式是一样的，只是目标改成了 `findLoadedClass` 函数。

新函数的实现如下：

```
HOOK_JNI(jobject, findLoadedClass, JNIEnv *env, jobject obj, jobject
class_loader, jstring name) {
    const char * nameC = env->GetStringUTFChars(name, JNI_FALSE);
//     ALOGD("findLoadedClass: %s", nameC);
    if (hideXposedClass) {
        if (strstr(nameC, "de/robv/android/xposed/") ||
            strstr(nameC, "me/weishu/epic") ||
            strstr(nameC, "me/weishu/exposed") ||
            strstr(nameC, "de.robv.android") ||
            strstr(nameC, "me.weishu.epic") ||
            strstr(nameC, "me.weishu.exposed")) {
            env->ReleaseStringUTFChars(name, nameC);
            return nullptr;
        }
    }
    jobject result = orig_findLoadedClass(env, obj, class_loader, name);
    env->ReleaseStringUTFChars(name, nameC);
    return result;
}
```

可以看到其实是在做一些过滤，把一些不希望被发现的 Class 过滤掉。

Binder Hook

同上，这次的目标是 getCallingUid 函数：

```
void BinderHook::init(JNIEnv *env) {
    const char *clazz = "android/os/Binder";
    JniHook::HookJniFun(env, clazz, "getCallingUid", "()I", (void *)
new_getCallingUid,
                        (void **) (&orig_getCallingUid), true);
}
```

最终会先调用一遍原生函数，然后再用下面这个函数做处理：

```
@Keep
public static int getCallingUid(int origCallingUid) {
  // 系统uid
  if (origCallingUid > 0 && origCallingUid < Process.FIRST_APPLICATION_UID)
return origCallingUid;
  // 非用户应用
  if (origCallingUid > Process.LAST_APPLICATION_UID) return origCallingUid;

  if (origCallingUid == SandBoxCore.getHostUid()) {
    return BActivityThread.getCallingBUid();
  }
  return origCallingUid;
}
```

如果是系统 UID 或不是用户应用的话就直接返回原本的值，而如果返回的结果是容器的 Uid ，那就要过滤一下替换成容器内目标应用的 Uid 了：

```
public static int getCallingBUid() {
  return getAppConfig() == null ? SandBoxCore.getHostUid() :
getAppConfig().callingBUid;
}
```

seccomp Hook

除此之位，Blackbox 还基于 seccomp 实现了对系统调用的 Hook：

```
void init_seccomp(JNIEnv *env, jclass clazz) {
    struct sock_filter filter[] = {
            BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data,
nr)),
            BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 0, 2),
            BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data,
args[4])),
            BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SECMAGIC, 0, 1),
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
            BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_TRAP)
    };

    struct sock_fprog prog;
    prog.filter = filter;
    prog.len = (unsigned short) (sizeof(filter) / sizeof(filter[0]));

    struct sigaction sa;
    sigset_t sigset;
    sigfillset(&sigset);
    sa.sa_sigaction = sig_callback;
    sa.sa_mask = sigset;
    sa.sa_flags = SA_SIGINFO;

    if (sigaction(SIGSYS, &sa, NULL) == -1) {
        return;
    }
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == -1) {
        return;
    }
    if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == -1) {
        return;
    }
    ALOGE("InitCvmSeccomp Successes");
}
```

触发上述的过滤条件时，会通过信号来回调：

```
void sig_callback(int signo, siginfo_t *info, void *data){
    int my_signo = info->si_signo;
    unsigned long syscall_number;
    unsigned long SYSARG_1, SYSARG_2, SYSARG_3, SYSARG_4, SYSARG_5, SYSARG_6;
#if defined(__aarch64__)
    syscall_number = ((ucontext_t *) data)->uc_mcontext.regs[8];
    SYSARG_1 = ((ucontext_t *) data)->uc_mcontext.regs[0];    SYSARG_2 =
((ucontext_t *) data)->uc_mcontext.regs[1];    SYSARG_3 = ((ucontext_t *)
data)->uc_mcontext.regs[2];    SYSARG_4 = ((ucontext_t *) data)-
>uc_mcontext.regs[3];    SYSARG_5 = ((ucontext_t *) data)-
>uc_mcontext.regs[4];    SYSARG_6 = ((ucontext_t *) data)-
>uc_mcontext.regs[5];#elif defined(__arm__)
    syscall_number = ((ucontext_t *) data)->uc_mcontext.arm_r7;
    SYSARG_1 = ((ucontext_t *) data)->uc_mcontext.arm_r0;
    SYSARG_2 = ((ucontext_t *) data)->uc_mcontext.arm_r1;
    SYSARG_3 = ((ucontext_t *) data)->uc_mcontext.arm_r2;
    SYSARG_4 = ((ucontext_t *) data)->uc_mcontext.arm_r3;
    SYSARG_5 = ((ucontext_t *) data)->uc_mcontext.arm_r4;
    SYSARG_6 = ((ucontext_t *) data)->uc_mcontext.arm_r5;
#else
#error "Unsupported architecture"
#endif
    switch (syscall_number) {
        case __NR_openat:{
            int fd = (int) SYSARG_1;
            const char *pathname = (const char *) SYSARG_2;
            int flags = (int) SYSARG_3;
            int mode = (int) SYSARG_4;
            ALOGE("测试%s",pathname);
#if defined(__aarch64__)
            ((ucontext_t *) data)->uc_mcontext.regs[0] = (uint64_t)fd;
            ((ucontext_t *) data)->uc_mcontext.regs[1] = (uint64_t)pathname;
((ucontext_t *) data)->uc_mcontext.regs[2] = (uint64_t)flags;
((ucontext_t *) data)->uc_mcontext.regs[3] = (uint64_t)mode;#elif
defined(__arm__)
            ((ucontext_t *) data)->uc_mcontext.arm_r0 = (uint32_t)fd;
            ((ucontext_t *) data)->uc_mcontext.arm_r1 = (uint32_t)pathname;
            ((ucontext_t *) data)->uc_mcontext.arm_r2 = (uint32_t)flags;
            ((ucontext_t *) data)->uc_mcontext.arm_r3 = (uint32_t)mode;
#endif
#if defined(__aarch64__)
            ((ucontext_t *) data)->uc_mcontext.regs[0] =
OriSyscall(__NR_openat, fd, (uint64_t)pathname, flags, mode, SECMAGIC,
SECMAGIC);
#elif defined(__arm__)
            ((ucontext_t *) data)->uc_mcontext.arm_r0 =
OriSyscall(__NR_openat, fd, (uint32_t)pathname, flags, mode, SECMAGIC,
SECMAGIC);
#endif
            break;
        }
        default:
            break;
    }
}
```

可以看到，在这类对 openat 的系统调用做了拦截，不过目前还并没有做什么过滤，未来应该会有更好的支持。

Xposed 支持

算是最后一个笔者好奇的点。笔者之前并没有使用过 Xposed 的经历，因此对相关的内容其实不是很了解。通过网上的一些资料了解到，Xposed 是通过修改 app_process 的方式向进程里注入代码实现的，而这些操作需要它能够对 zygote 实现 Hook，那么对于 VirtualApp 来说要如何支持呢？

入口从创建应用时的 `newApplication` 开始：

```
@Override
public Application newApplication(ClassLoader cl, String className, Context
context)
    throws InstantiationException, IllegalAccessException,
ClassNotFoundException {
  ContextCompat.fix(context);
  fixSharedLibraryLoaders(cl);
  BActivityThread.currentActivityThread().loadXposed(context);
  delegateAppClassLoader = context.getClassLoader();
  if (RomUtil.isHuawei()) {
    hookHwEnvironment$UserEnvironment();
  }
  return super.newApplication(cl, className, context);
}
```

因为之前创建启动 `Activity` 时已经用 `AppInstrumentation` 替换掉了 `Instrumentation` ，而在正常的创建流程中会调用该对象的 `newApplication` 函数，而 Xposed 的注入就发生在这个时机。这个时候应用对应的进程已经创建好了，后续就是等待绑定的流程了，而应用还尚且没有开始运行，因此算是合适的时机之一。

可以看到代码中在此刻加载了 Xposed，然后再带调用原生的 `newApplication` 恢复创建流程：

```
public void loadXposed(Context context) {
  String vPackageName = getAppPackageName();
  String vProcessName = getAppProcessName();
  if (!TextUtils.isEmpty(vPackageName)
      && !TextUtils.isEmpty(vProcessName)
      && BXposedManager.get().isXPEnable()) {
    assert vPackageName != null;
    assert vProcessName != null;

    boolean isFirstApplication = vPackageName.equals(vProcessName);

    List<InstalledModule> installedModules =
BXposedManager.get().getInstalledModules();
    for (InstalledModule installedModule : installedModules) {
      if (!installedModule.enable) {
        continue;
      }
      try {
        PineXposed.loadModule(new
File(installedModule.getApplication().sourceDir));
      } catch (Throwable e) {
        e.printStackTrace();
      }
    }
    try {
      PineXposed.onPackageLoad(
          vPackageName,
          vProcessName,
          context.getApplicationInfo(),
          isFirstApplication,
          context.getClassLoader());
    } catch (Throwable ignored) {
    }
  }
  if (SandBoxCore.get().isHideXposed()) {
    NativeCore.hideXposed();
  }
}
```

代码中遍历了容器内安装的所有模块，并通过 `PineXposed.loadModule` 来加载，这似乎是一个框架的内部实现，翻到了作者的原文笔记。里面提到的说是直接套了 SandHook 的 API ， 继续往下翻似乎就是 SandVXposed ， 不过最后也没咋找到我想看的，于是只好自己翻起其他的资料对照着源码试着啃啃看了。

首先是加载 Xposed 的相关代码：

```
    final String filename = "assets/xposed_init";
    if (mcl instanceof ModuleClassLoader) {
      // Fast and provided more error info
      URL url = ((ModuleClassLoader) mcl).findResource(filename);
      initIs = url != null ? url.openStream() : null;
    } else {
      initIs = mcl.getResourceAsStream(filename);
    }
    if (initIs == null) {
      Log.e(TAG, "  Failed to load module " + modulePath);
      Log.e(TAG, "  assets/xposed_init not found in the module APK");
      return;
    }
  } catch (IOException e) {
```

```
  Log.e(TAG, "  Failed to load module " + modulePath);
  Log.e(TAG, "  Cannot open assets/xposed_init in the module APK", e);
  return;
}
  BufferedReader xposedInitReader = new BufferedReader(new
InputStreamReader(initIs));
```

然后从资源里读取代码然后加载:

```
Class<?> c = mcl.loadClass(className);
```

然后下面有一个初始化 Xposed 的操作:

```
((IXposedHookZygoteInit) callback).initZygote(param);
```

起初还以为是要注入 Zygote ，但是没有 root 也不能注入才对，翻了下源代码似乎是这样的:

```
    @Override
    public void initZygote(StartupParam startupParam) throws Throwable {
        hookXposedFrameworkApi(startupParam);
    }

    /**
     * 用于分析第三方 XP 模块采用了哪些钩子，主要用于模块包体比较大，
     * 做了加固或者混淆做得比较彻底的场景
     * <p>
     * 钩子的目标类和方法名，可在 Xposed Installer 的日志中查看
     */
    private void hookXposedFrameworkApi(StartupParam param) {
        hookXpFindAndHookMethod();
        hookXpFindAndHookConstructor();
        hookXpHookAllMethods();
        hookXpHookAllConstructors();
        }
    private void hookXpFindAndHookMethod() {
      try {
        Class cls = XposedHelpers.findClass(
                "de.robv.android.xposed.XposedHelpers",
                HookUtils.class.getClassLoader());
        XposedHelpers.findAndHookMethod(cls, "findAndHookMethod",
                Class.class, String.class, Object[].class, new XC_MethodHook()
{
                    @Override
                    protected void beforeHookedMethod(MethodHookParam param)
```

```java
throws Throwable {
                    super.beforeHookedMethod(param);
                    XposedBridge.log("findAndHookMethod: cls = " +
param.args[0]
                            + ", method = " + param.args[1]);
                  }
                });
      } catch (Exception e) {
        XposedBridge.log(e);
      }
    }

    private void hookXpFindAndHookConstructor() {
      try {
        Class cls = XposedHelpers.findClass(
                "de.robv.android.xposed.XposedHelpers",
                HookUtils.class.getClassLoader());
        XposedHelpers.findAndHookMethod(cls, "findAndHookConstructor",
                Class.class, Object[].class, new XC_MethodHook() {
                  @Override
                  protected void beforeHookedMethod(MethodHookParam param)
throws Throwable {
                    super.beforeHookedMethod(param);
                    XposedBridge.log("findAndHookConstructor: cls = " +
param.args[0]);
                  }
                });
      } catch (Exception e) {
        XposedBridge.log(e);
      }
    }

    private void hookXpHookAllMethods() {
      try {
        Class cls = XposedHelpers.findClass(
                "de.robv.android.xposed.XposedBridge",
                HookUtils.class.getClassLoader());
        Class clsXC_MethodHook = XposedHelpers.findClass(
                "de.robv.android.xposed.XC_MethodHook",
HookUtils.class.getClassLoader()
        );
        XposedHelpers.findAndHookMethod(cls, "hookAllMethods",
                Class.class, String.class, clsXC_MethodHook, new
XC_MethodHook() {
                  @Override
                  protected void beforeHookedMethod(MethodHookParam param)
```

```
throws Throwable {
                    super.beforeHookedMethod(param);
                    XposedBridge.log("hookAllMethods: cls = " + param.args[0]
                            + ", method = " + param.args[1]);
                  }
                });
      } catch (Exception e) {
        XposedBridge.log(e);
      }
    }

    private void hookXpHookAllConstructors() {
      try {
        Class cls = XposedHelpers.findClass(
                "de.robv.android.xposed.XposedBridge",
                HookUtils.class.getClassLoader());
        Class clsXC_MethodHook = XposedHelpers.findClass(
                "de.robv.android.xposed.XC_MethodHook",
HookUtils.class.getClassLoader()
        );
        XposedHelpers.findAndHookMethod(cls, "hookAllConstructors",
                Class.class, clsXC_MethodHook, new XC_MethodHook() {
                  @Override
                  protected void beforeHookedMethod(MethodHookParam param)
throws Throwable {
                    super.beforeHookedMethod(param);
                    XposedBridge.log("hookAllConstructors: cls = " +
param.args[0]);
                  }
                });
      } catch (Exception e) {
        XposedBridge.log(e);
      }
    }
```

看起来似乎不是注入 Zygote ，只是对其中的一些方法做了 Hook，既然如此就不需要真的去注入 Zygote 了，只需要在加载应用时把调一下这个方法把它们拦截下来就行了。

完成 Hook 以后再加载对应模块：

```
hookLoadPackage((IXposedHookLoadPackage) callback);

public static void hookLoadPackage(IXposedHookLoadPackage callback) {
  sLoadedPackageCallbacks.add(new XC_LoadPackage.Wrapper(callback));
}
```

这个 `sLoadedPackageCallbacks` 只是暂存。在对每个模块做完上述操作以后，下面还有一段：

```
try {
  PineXposed.onPackageLoad(
      vPackageName,
      vProcessName,
      context.getApplicationInfo(),
      isFirstApplication,
      context.getClassLoader());
} catch (Throwable ignored) {
}
```

往下跟一下：

```
public static void onPackageLoad(
    String packageName,
    String processName,
    ApplicationInfo appInfo,
    boolean isFirstApp,
    ClassLoader classLoader) {
  XC_LoadPackage.LoadPackageParam param =
      new XC_LoadPackage.LoadPackageParam(sLoadedPackageCallbacks);
  param.packageName = packageName;
  param.processName = processName;
  param.appInfo = appInfo;
  param.isFirstApplication = isFirstApp;
  param.classLoader = classLoader;
  XC_LoadPackage.callAll(param);
}

/** @hide */
public static void callAll(Param param) {
  if (param.callbacks == null)
    throw new IllegalStateException("This object was not created for use with
callAll");

  for (int i = 0; i < param.callbacks.length; i++) {
    try {
      ((XCallback) param.callbacks[i]).call(param);
    } catch (Throwable t) {
      XposedBridge.log(t);
    }
  }
}
```

这里最后会去调用 `XC_LoadPackage` 下的方法：

```
/** @hide */
@Override
protected void call(Param param) throws Throwable {
  if (param instanceof LoadPackageParam) handleLoadPackage((LoadPackageParam)
param);
}
```

最后这个 `handleLoadPackage` 方法就会调用到模块里那个重载后的 `handleLoadPackage` 来实现模块功能的加载了。

不过这里有一点很奇怪的是，为什么需要每个模块都调用一遍 `initZygote` 呢？岂不是在重复 Hook 那些方法么？看起来似乎每次都把模块信息传进去了，但是好像没有用到这个信息，那又为什么需要每次都调用一遍呢？

没想明白，如果有大佬知道的话还请多多指教。

重定向 Hook 总结和疑问

在最后一部分我们可以看到，如果通过 Seccomp 对 open 等系统调用做拦截的话，应该上层的很多重定向都可以放掉了？毕竟不管上层用了什么函数，底层对文件的访问都是需要系统调用去支持的，从这个层面说，只要在系统调用层面把路径全部做好过滤，是不是上层的重定向可以直接删掉？

不过 Seccomp 这一策略首先支持的版本比较新，老设备肯定是没有的，以及另一方面是，检测 Seccomp 是不是较为简单呢？笔者目前对这些问题尚没有答案，还需要请教各位大佬。

参考文章

https://blog.canyie.top/2020/04/27/dynamic-hooking-framework-on-art/
https://wufengxue.github.io/2019/11/01/get-3rd-xp-module-hookers.html
https://blog.canyie.top/2020/02/03/a-new-xposed-style-framework/