Android 应用多开对抗实践

白天午睡梦到些以前给某行业安全峰会写了材料，醒来后把记得的部分重新整理一下，分享出来给大家，尽量写得简洁明了。

未必准确，仅供参考，欢迎纠正补充

应用多开技术总结

系统级技术

多开技术方案	发行版 APP
Android 多用户功能	OEM系统自带的"手机分身"、"应用双开"，和 "Island/炼妖壶" 等各种 "Android for work" 产品
`chroot` / `lxc`	暂无发现 APP

- 多用户功能

  多用户模式主要用到 `UserManager` 相关类，切换不同的用户，在不同的用户下运行 App，实现多开。最直观的例子是 Android 手机上的 `多用户` 功能， `手机分身` 功能，以及 `am switch-user` 命令，这种简单粗暴的用法会将 Android 服务都运行一份，如果只用于应用多开，且不说资源消耗，切换用户是在麻烦。

  在 Android 5.0 在基于多用户功能 添加了 `Android for work` 功能，可以在同一个桌面启动器下使用受限用户启动 APP，不再需要切换界面。同时将权限开发给了非系统应用。

- chroot

  UNIX 的 chroot 系统调用在 Android 上也能用，需要 root 权限。在本地挂载运行精简版系统镜像，使用远程桌面软件如 VNC 等访问本地多开的系统。尚未发现发行版 APP，可能在 ARM 服务器云手机中用到。

用户级技术

多开技术方案	发行版 APP
VirtualApp	VirtualXposed, DualSpace
MultiDroid	LBE平行空间, Parallel Space
DroidPlugin	分身大师
Excelliance	双开助手, MultiAccount
其它	虚拟大师

在用户级的多开技术中，还可以在按设计用途划分出三类

- “容器”：VirtualApp、MultiDroid
- 热更新/插件化：DroidPlugin、Excelliance
- 虚拟系统：虚拟大师

具体实现原理大家可以翻论坛里的 精品贴，这里不多描述。

值得一提的是，某云手机团队的 "虚拟大师" 产品，实现在用户态运行了一个精简版的 Android 系统镜像，在系统库中拦截了几乎所有系统调用，使用类似前文提到的 `chroot` 挂载系统镜像的方法运行，有兴趣的同学可以看一看。

拆招

反系统级应用多开

> 仅多用户方案的多开，忽略 `chroot` / `lxc`

简单粗暴的代码

```
// --- C++ ---
#include <unistd.h>
bool isDualApp(){return 0 != getuid()/100000;}
```

```
// --- Java ---
import android.os.Process;
static boolean isDualApp(){return 0 != Process.myUid() / 100000;}
```

一行代码完事了?

完事了，真的完事了。

但是为什么?

Android 系统中，如果开启了多用户模式，会存在一个主用户和若干受限用户。

> 把 MIUI 的 "手机分身" 和 "应用双开" 功能都打开，可以看到有三用户，0、11 和 999，分别对应主用户、"手机分身" 和 "应用双开"

```
# --- adb shell ---
$ ls -al /data/user/
total 52
drwx--x--x   4 system system  4096 2019-09-05 11:49 .
drwxrwx--x  42 system system  4096 2019-04-22 20:32 ..
lrwxrwxrwx   1 root   root      10 1970-08-23 18:57 0 -> /data/data
drwxrwx--x 221 system system 16384 2019-09-05 11:50 11
drwxrwx--x  13 system system 16384 2019-09-12 17:53 999
```

使用多用户模式实现的多开，在客户端中可以通过 Android SDK 的 `UserManger` 类判断当前运行 APP 的用户是否为主用户和受限用户

```
// android.os.UserManger.java
public boolean isAdminUser() {
    return isUserAdmin(UserHandle.myUserId());
}
// ...
public boolean isPrimaryUser() {
    UserInfo user = getUserInfo(UserHandle.myUserId());
    return user != null && user.isPrimary();
}
```

顺着线索，找到 `UserHandler` 类

```
// android.os.UserHandle.java
@Deprecated
/**
* @hide A user id constant to indicate the "owner" user of the device
* @deprecated Consider using either {@link UserHandle#USER_SYSTEM} constant or
* check the target user's flag {@link android.content.pm.UserInfo#isAdmin}.
*/
@Deprecated
public static final @UserIdInt int USER_OWNER = 0;
// ...
public static final @UserIdInt int USER_SYSTEM = 0;
// ...
public static final int PER_USER_RANGE = 100000;
// ...
public static @UserIdInt int getUserId(int uid) {
    if (MU_ENABLED) {
        return uid / PER_USER_RANGE;
    } else {
        return UserHandle.USER_SYSTEM;
    }
}
```

可看到通过 uid/100000 获得 Android 的 UserId，同时通过其它 final 字段和相关注释得知 `OWNER/SYSTEM/ADMIN` 的 UserId 是 0，因此我们可以通过 uid/100000 为 0 判断为主用户，非主用户直接判为多开即可。

验证

使用上文提到的 MIUI 中的 "应用双开" 功能，在进程中找到 UserId 999 运行的进程，因为第一列显示成了用户名，进 `/proc/${PID}/status` 查看进程 uid。uid / 100000 是 999，没毛病。

```
# --- adb shell ---
$ ps -ef | grep u999
u999_a118    14392   905 0 14:07:52 ?     00:00:01 com.miui.analytics
u999_system  19793   905 0 19:55:24 ?     00:00:00 com.android.keychain
shell        20408 13712 3 19:58:11 pts/0 00:00:00 grep u999
$ cat /proc/14392/status
Name:   .miui.analytics
Umask:  0077
State:  S (sleeping)
Tgid:   14392
Ngid:   0
Pid:    14392
PPid:   905
TracerPid:      0
Uid:    99910118        99910118        99910118        99910118
Gid:    99910118        99910118        99910118        99910118
```

可改进

- 区分各种系统级双开/分身模式

反用户级应用多开

仍然是简单粗暴的代码

测试代码一

```
// --- C++ ---
#include <unistd.h>
#include <sys/stat.h>
#include <string>
bool isDualApp(std::string dataDir) {
    return 0 == access((dataDir + "/../").c_str(), R_OK);
}
```

测试代码二

```
// --- Java ---
import java.io.File;
boolean isDualApp(String dataDir){
    return new File(dataDir + File.separator + "..").canRead();
}
```

dataDir 目录的父级目录理论上归属 `system` 用户或 `root` 用户，除非是系统应用，否则不能访问。

> **补充**
>
> 为什么使用 dataDir 而不是 nativeLibraryDir 或 sourceDir 呢?
>
> 看 nativeLibraryDir 和 sourceDir 目录所在的位置，base.apk 权限 `rw-r-r--r--`任意用户读取，lib 目录权限 `rwxr-xr-x` 任意用户读取和执行

```
# ls -al /data/app/com.example.checksandbox-ElZnrZIb5m2rbBv_3K8nZQ\=\=/
total 2420
drwxr-xr-x  3 system system    4096 2019-09-24 16:15 .
drwxrwx--x 80 system system   12288 2019-09-24 16:15 ..
-rw-r--r--  1 system system 2440234 2019-09-24 16:15 base.apk
drwxr-xr-x  3 system system    4096 2019-09-24 16:15 lib
```

而 dataDir 的目录权限 `rwx------`，私有不公开

```
# ls -al /data/user/0/com.example.checksandbox/
total 40
drwx------   5 u0_a366 u0_a366        4096 2019-09-24 16:15 .
drwxrwx--x 271 system  system        20480 2019-09-24 16:15 ..
drwxrws--x   2 u0_a366 u0_a366_cache  4096 2019-09-24 16:15 cache
drwxrws--x   2 u0_a366 u0_a366_cache  4096 2019-09-24 16:15 code_cache
drwxrwxr-x   5 u0_a366 u0_a366        4096 2019-09-24 16:15 lldb
```

因此，对于用户级多开软件来说，仅是实现多开功能，完全可以复用原本的 base.apk 和 lib，不需要在自己目录中释放一份，而 dataDir 目录却是必须要另外准备的。

验证

代码放 APP 里跑一遍看看

测试代码一结果

技术	发行产品	结果
DroidPlugin	分身大师	**阴性**
VirtualApp	VirtualXposed	阳性 +
VirtualApp	DualSpace	阳性 +
MultiDroid	LBE平行空间	阳性 +
Excelliance	双开助手	阳性 +
其它	虚拟大师	阳性 +

emmm，看来三六零在文件系统上的功能做得挺完善的，`access` 访问 dataDir 的父目录时返回了 -1 (errno: 13 Permission denied)，其它的没什么问题

测试代码二结果

技术	发行产品	结果
DroidPlugin	分身大师	阳性 +
VirtualApp	VirtualXposed	阳性 +
VirtualApp	DualSpace	阳性 +
MultiDroid	LBE平行空间	阳性 +
Excelliance	双开助手	阳性 +
其它	虚拟大师	阳性 +

到底还是高估了三六零？？？

威力加强版

上面给的两份代码容易被绕过，稍微换个姿势，获取到真实的路径，再利用父级目录进行检测。

由于睡懵了忘记了细节，这里用 Java 写了份 "伪" 代码，由于没能绕过 hook，多少会存在瑕疵，但是原本的 ASM 代码绕过各种 Hook，达到通杀

```
// --- Java ---
import android.support.annotation.RequiresApi;
import java.io.File;
import java.io.FileDescriptor;
import java.io.FileOutputStream;
import java.lang.reflect.Field;
import java.nio.file.Files;
import java.nio.file.Paths;

@RequiresApi(api = Build.VERSION_CODES.O)
boolean isDualAppEx(String dataDir) {
    try {
        String simpleName = "wtf_jack";
        String testPath = dataDir + File.separator + simpleName;
        FileOutputStream fos = new FileOutputStream(testPath);
        FileDescriptor fileDescriptor = fos.getFD();
        Field fid_descriptor =
fileDescriptor.getClass().getDeclaredField("descriptor");
        fid_descriptor.setAccessible(true);
        // 获取到 fd
        int fd = (Integer) fid_descriptor.get(fileDescriptor);
        // fd 反查真实路径
        String fdPath = String.format("/proc/self/fd/%d", fd);
        String realPath = Files.readSymbolicLink(Paths.get(fdPath)).toString();
        if (!realPath.substring(realPath.lastIndexOf(File.separator))
```

```
                .equals(File.separator + simpleName)){
                // 文件名被修改
                return true;
            }
            // 尝试访问真实路径的父目录
            String fatherDirPath = realPath.replace(simpleName, "..");
            Log.d(TAG, "isDualAppEx: " + fatherDirPath);
            File fatherDir = new File(fatherDirPath);
            if (fatherDir.canRead()) {
                // 父目录可访问
                return true;
            }
        } catch (Exception e) {
            e.printStackTrace();
            return true;
        }
        return false;
    }
```

对用户级应用多开的理解

两个关键技术点：Binder IPC 拦截 、系统函数拦截

两个直观特征：沙盒破损、共享 UID

Binder IPC 拦截用于重定向 APP 与 Android 系统服务、其它应用的远程调用；

系统函数拦截主要用于重定向 I/O 操作，如果搞定所有 libc 函数，再配合修改过的 linker ，可以在一定程度上仿真 Linux 系统；

在用户级的多开中，没有系统的支持，缺少沙盒机制，应用的数据目录一般内嵌在宿主的数据目录中。同时，在同一个用户级多开软件中，内部的 APP 共享宿主的 UID，意味着他们都具有相同的权限，在系统层面上可以相互访问。因此，围绕这点，除了上文给出的方法外，还有三百六十五天不重复的姿势能够检出用户级的多开。

> - 由于是在用户态实现的隔离，不建议用于政企办公保密场景，有此类需求的可以直接用系统自带的 "Android for work" 功能。

业务前端白名单

如果己方有用户级的多开辅助工具，需要在 **前端** 检测上添加白名单，通常有三种白名单：

- APK 包名
- APK 签名公钥
- APK 内文件(文件名/CRC/HASH)

前端白名单会成为一个很好的攻击点，需要相关业务的后端一起配合来弥补

后记

不要觉得各种 “奔放稳定框架” 稳，只是厂商在权衡 KPI 之后的围三阙一，老大哥可以用各种姿势虎摸狗头。

附件提供 demo，欢迎把玩，欢迎补充。

> PS: 看雪的 markdown 不支持 mermaid，由不想贴图，讲究看看吧