Android进程保活

。**坦白的讲，我是真心不希望有这种黑科技存在的，它只会滋生更多的流氓应用，拖垮我大 Android 平台的流畅性。**

扯了这么多，接下来就直接进入本文的正题，谈谈关于进程保活的知识。提前声明以下四点

* **本文是本人开发 Android 至今综合各方资料所得**
* **不以节能来维持进程保活的手段，都是耍流氓**
* **本文不是教你做永生不死的进程，如果指望实现进程永生不死，请忽略本文**
* **本文有错误的地方，欢迎留下评论互相探讨（拍砖请轻拍）**

**保活手段**

当前业界的Android进程保活手段主要分为**黑、白、灰**三种，其大致的实现思路如下：

**黑色保活**：不同的app进程，用广播相互唤醒（包括利用系统提供的广播进行唤醒）

**白色保活**：启动前台Service

**灰色保活**：利用**系统的漏洞**启动前台Service

**黑色保活**

所谓黑色保活，就是利用不同的app进程使用广播来进行相互唤醒。举个3个比较常见的场景：

**场景1**：开机，网络切换、拍照、拍视频时候，利用系统产生的广播唤醒app

**场景2**：接入第三方SDK也会唤醒相应的app进程，如微信sdk会唤醒微信，支付宝sdk会唤醒支付宝。由此发散开去，就会直接触发了下面的 **场景3**

**场景3**：假如你手机里装了支付宝、淘宝、天猫、UC等阿里系的app，那么你打开任意一个阿里系的app后，有可能就顺便把其他阿里系的app给唤醒了。（只是拿阿里打个比方，其实BAT系都差不多）

**没错，我们的Android手机就是一步一步的被上面这些场景给拖卡机的。**

针对**场景1**，估计Google已经开始意识到这些问题，所以在最新的Android N取消了 ACTION\_NEW\_PICTURE（拍照），ACTION\_NEW\_VIDEO（拍视频），CONNECTIVITY\_ACTION（网络切换）等三种广播，无疑给了很多app沉重的打击。我猜他们的心情是下面这样的



而开机广播的话，记得有一些定制ROM的厂商早已经将其去掉。

针对**场景2**和**场景3**，因为调用SDK唤醒app进程属于正常行为，此处不讨论。但是在借助LBE分析app之间的唤醒路径的时候，发现了两个问题：

1. 很多推送SDK也存在唤醒app的功能
2. app之间的唤醒路径真是多，且错综复杂

我把自己使用的手机测试结果给大家围观一下（**我的手机是小米4C，刷了原生的Android5.1系统，且已经获得Root权限才能查看这些唤醒路径**）



15组相互唤醒路径



全部唤醒路径

我们直接点开 **简书** 的唤醒路径进行查看



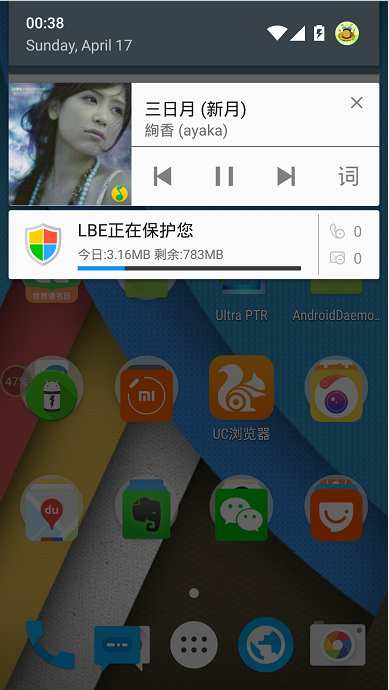
简书唤醒路径

可以看到以上3条唤醒路径，但是涵盖的唤醒应用总数却达到了23+43+28款，数目真心惊人。请注意，这只是我手机上一款app的唤醒路径而已，到了这里是不是有点细思极恐。

当然，这里依然存在一个疑问，就是LBE分析这些唤醒路径和互相唤醒的应用是基于什么思路，我们不得而知。所以我们也无法确定其分析结果是否准确，如果有LBE的童鞋看到此文章，不知可否告知一下思路呢？但是，手机打开一个app就唤醒一大批，我自己可是亲身体验到这种酸爽的......

**白色保活**

白色保活手段非常简单，就是调用系统api启动一个前台的Service进程，这样会在系统的通知栏生成一个Notification，用来让用户知道有这样一个app在运行着，哪怕当前的app退到了后台。如下方的LBE和QQ音乐这样：



**灰色保活**

灰色保活，这种保活手段是应用范围最广泛。它是利用系统的漏洞来启动一个前台的Service进程，与普通的启动方式区别在于，它不会在系统通知栏处出现一个Notification，看起来就如同运行着一个后台Service进程一样。这样做带来的好处就是，用户无法察觉到你运行着一个前台进程（因为看不到Notification）,但你的进程优先级又是高于普通后台进程的。那么如何利用系统的漏洞呢，大致的实现思路和代码如下：

* 思路一：API < 18，启动前台Service时直接传入new Notification()；
* 思路二：API >= 18，同时启动两个id相同的前台Service，然后再将后启动的Service做stop处理；

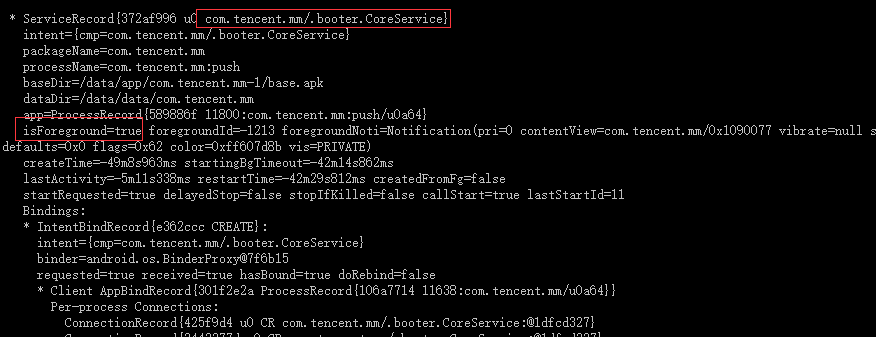
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class GrayService extends Service {        private final static int GRAY\_SERVICE\_ID = 1001;        @Override      public int onStartCommand(Intent intent, int flags, int startId) {          if (Build.VERSION.SDK\_INT < 18) {              startForeground(GRAY\_SERVICE\_ID, new Notification());//API < 18 ，此方法能有效隐藏Notification上的图标          } else {              Intent innerIntent = new Intent(this, GrayInnerService.class);              startService(innerIntent);              startForeground(GRAY\_SERVICE\_ID, new Notification());          }            return super.onStartCommand(intent, flags, startId);      }        ...      ...        /\*\*       \* 给 API >= 18 的平台上用的灰色保活手段       \*/      public static class GrayInnerService extends Service {            @Override          public int onStartCommand(Intent intent, int flags, int startId) {              startForeground(GRAY\_SERVICE\_ID, new Notification());              stopForeground(true);              stopSelf();              return super.onStartCommand(intent, flags, startId);          }        }  } |

代码大致就是这样，能让你神不知鬼不觉的启动着一个前台Service。其实市面上很多app都用着这种灰色保活的手段，什么？你不信？好吧，我们来验证一下。流程很简单，打开一个app，看下系统通知栏有没有一个 Notification，如果没有，我们就进入手机的adb shell模式，然后输入下面的shell命令

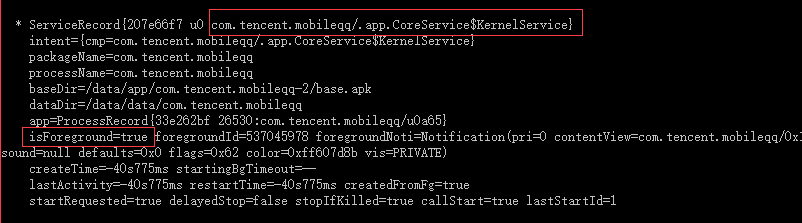
|  |  |
| --- | --- |
| 1 | dumpsys activity services PackageName |

打印出指定包名的所有进程中的Service信息，看下有没有 **isForeground=true** 的关键信息。如果通知栏没有看到属于app的 Notification 且又看到 **isForeground=true** 则说明了，此app利用了这种灰色保活的手段。

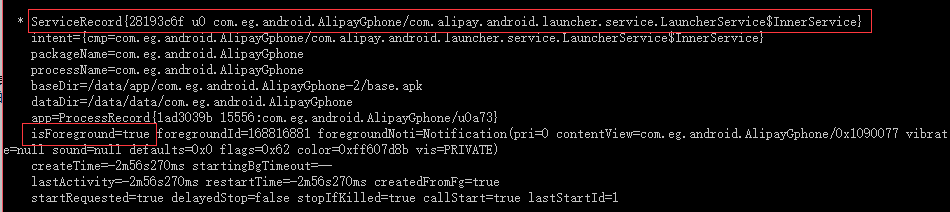
下面分别是我手机上微信、qq、支付宝、陌陌的测试结果，大家有兴趣也可以自己验证一下。



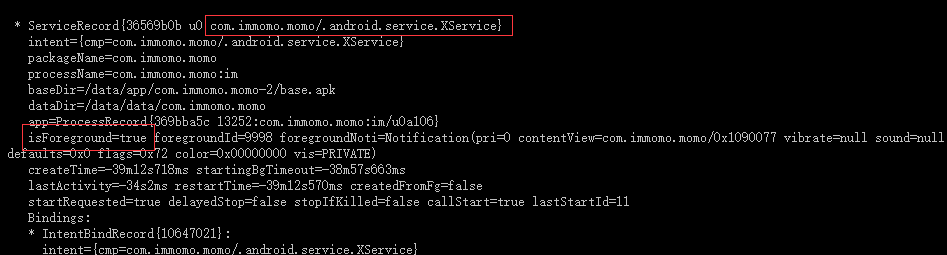
微信



手Q

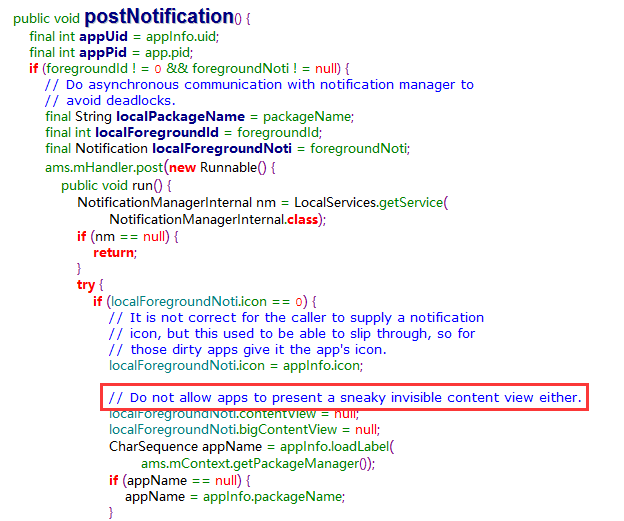


支付宝



陌陌

其实Google察觉到了此漏洞的存在，并逐步进行封堵。这就是为什么这种保活方式分 API >= 18 和 API < 18 两种情况，从Android5.0的ServiceRecord类的postNotification函数源代码中可以看到这样的一行注释



当某一天 API >= 18 的方案也失效的时候，我们就又要另谋出路了。需要注意的是，**使用灰色保活并不代表着你的Service就永生不死了，只能说是提高了进程的优先级。如果你的app进程占用了大量的内存，按照回收进程的策略，同样会干掉你的app。**感兴趣于灰色保活是如何利用系统漏洞不显示 Notification 的童鞋，可以研究一下系统的 ServiceRecord、NotificationManagerService 等相关源代码，因为不是本文的重点，所以不做详述。

**唠叨的分割线**

到这里基本就介绍完了**黑、白、灰**三种实现方式，仅仅从代码层面去讲保活是不够的，我希望能够通过系统的进程回收机制来理解保活，这样能够让我们更好的避免踩到进程被杀的坑。

**进程回收机制**

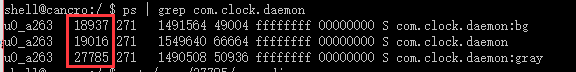
熟悉Android系统的童鞋都知道，系统出于体验和性能上的考虑，app在退到后台时系统并不会真正的kill掉这个进程，而是将其缓存起来。打开的应用越多，后台缓存的进程也越多。在系统内存不足的情况下，系统开始依据自身的一套进程回收机制来判断要kill掉哪些进程，以腾出内存来供给需要的app。这套杀进程回收内存的机制就叫 **Low Memory Killer** ，它是基于Linux内核的 **OOM Killer（Out-Of-Memory killer）**机制诞生。

了解完 **Low Memory Killer**，再科普一下**oom\_adj**。什么是**oom\_adj**？它是linux内核分配给每个系统进程的一个值，代表进程的优先级，进程回收机制就是根据这个优先级来决定是否进行回收。对于**oom\_adj**的作用，你只需要记住以下几点即可：

* **进程的oom\_adj越大，表示此进程优先级越低，越容易被杀回收；越小，表示进程优先级越高，越不容易被杀回收**
* **普通app进程的oom\_adj>=0,系统进程的oom\_adj才可能<0**

那么我们如何查看进程的**oom\_adj**值呢，需要用到下面的两个shell命令

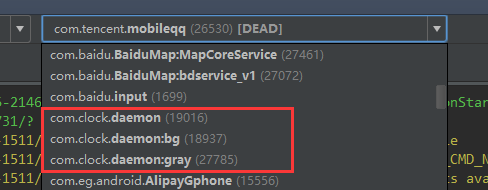
|  |  |
| --- | --- |
| 1 | ps | grep PackageName //获取你指定的进程信息 |



这里是以我写的demo代码为例子，红色圈中部分别为下面三个进程的ID

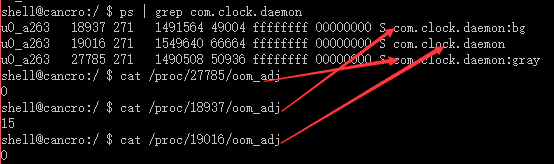
UI进程：**com.clock.daemon**  
普通后台进程：**com.clock.daemon:bg**  
灰色保活进程：**com.clock.daemon:gray**

当然，这些进程的id也可以通过AndroidStudio获得

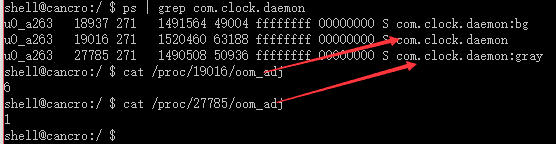


接着我们来再来获取三个进程的**oom\_adj**

|  |  |
| --- | --- |
| 1 | cat /proc/进程ID/oom\_adj |



从上图可以看到UI进程和灰色保活Service进程的**oom\_adj=0**，而普通后台进程**oom\_adj=15**。到这里估计你也能明白，**为什么普通的后台进程容易被回收，而前台进程则不容易被回收了吧。**但明白这个还不够，接着看下图



上面是我把app切换到后台，再进行一次**oom\_adj**的检验，你会发现UI进程的值从0变成了6,而灰色保活的Service进程则从0变成了1。这里可以观察到，**app退到后台时，其所有的进程优先级都会降低。但是UI进程是降低最为明显的，因为它占用的内存资源最多，系统内存不足的时候肯定优先杀这些占用内存高的进程来腾出资源。所以，为了尽量避免后台UI进程被杀，需要尽可能的释放一些不用的资源，尤其是图片、音视频之类的**。

从Android官方文档中，我们也能看到优先级从高到低列出了这些不同类型的进程：**Foreground process**、**Visible process**、**Service process**、**Background process**、**Empty process**。而这些进程的oom\_adj分别是多少，又是如何挂钩起来的呢？推荐大家阅读下面这篇文章：

<http://www.cnblogs.com/angeldevil/archive/2013/05/21/3090872.html>

**总结**

絮絮叨叨写完了这么多，最后来做个小小的总结。回归到开篇提到QQ进程不死的问题，我也曾认为存在这样一种技术。可惜我把手机root后，杀掉QQ进程之后就再也起不来了。有些手机厂商把这些知名的app放入了自己的白名单中，保证了进程不死来提高用户体验（如微信、QQ、陌陌都在小米的白名单中）。如果从白名单中移除，他们终究还是和普通app一样躲避不了被杀的命运，为了尽量避免被杀，还是老老实实去做好优化工作吧。

**所以，进程保活的根本方案终究还是回到了性能优化上，进程永生不死终究是个彻头彻尾的伪命题！**

总结：怎么进程保活

1. 在不清理进程的情况下
   1. 提高进程的优先级

共5中 https://www.jianshu.com/p/df766d5573a5

根据进程的优先级，可以将进行划分为5级：  
1、前台进程（Foreground process）：用户当前操作所必需的进程。在给定时间内，前台进程同时存在的数量并不多。只有在内存不足以支持它们同时存在时，系统才会终止它们。例如： A、拥有用户正在交互的Activity（已调用onResume） C、拥有正在“前台”运行的Service（服务已调用startForeground）

2、可见进程（Visible process）: 没有任何前台组件，但仍会影响用户在屏幕上所见内容的进程  
3、服务进程（Service process）: A、使用startService启动的服务，并且不属于上述两个更高级别的进程。  
4、后台进程（Background process）后台进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存给前台进程、可见进程和服务进程  
5、空进程（Empty process）A、不含任何活动的应用组件的进程

提高进程的优先级

1.设置只有一个像素的Activity，

2.使用前台服务，同时把关闭通知，不过这个在7.0以后以及修复了

思路一：API < 18，启动前台Service时直接传入new Notification()；

思路二：API >= 18，同时启动两个id相同的前台Service，然后再将后启动的Service做stop处理

用其他进程去唤醒

1. 使用双进程守护，开启两个服务，指定当一个服务被杀死是，onDestroy中通过AIdL通知另外一个进程的服务，在另外一个进程中 重新启动服务

补充：对process属性的设置有两种形式：

        第一种形式如 android:process=":remote"，以冒号开头，冒号后面的字符串原则上是可以随意指定的。如果我们的包名为“com.example.processtest”，则实际的进程名为“com.example.processtest:remote”。这种设置形式表示该进程为当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中。

第二种情况如 android:process="com.example.processtest.remote"，以小写字母开头，表示运行在一个以这个名字命名的全局进程中，其他应用通过设置相同的ShareUID可以和它跑在同一个进程。

1. 利用静态注册广播，监听开机或者电量改变时的的广播，在Android8.0以后对广播进行了限制

首先记住两点：显式广播和隐式广播属于发送广播行为；静态注册和动态注册属于注册广播接收器行为，

  Google为了延长电池的使用（防止多个app响应造成的问题），同时避免资源的浪费，便限制静态注册的广播接收器无法接收到隐式 ，但是能接收到显式广播，动态注册的广播接收器可以接收任意广播

Intent类型

    显式Intent：按名称（完全限定类名）指定到启动的组件。如 Intent intent = new Intent(this, MyReceive.class)，通常，在自己的应用中使用显式Intent来启动组件，这是因为知道要启动的Activity或服务的类名。例如，启动新Activity以响应用户操作，或者启动服务以在后台下载

    隐式Intent：不会指定特定的组件，而是声明要执行的常规操作，从而允许其他应用中的组件来处理它。如 Intent intent = new Intent(Intent.ACTION\_VIEW)；如需在地图上向用户显示位置，则可以使用隐式Intent，请求另一具有此功能的应用在地图上显示指定的位置

发送显式Intent的广播就是显式广播，发送隐式Intent的广播就是隐式广播

这个android:sharedUserId属性不只可以把apk放到系统进程中，也可以配置多个APK运行在一个进程中，这样可以共享数据，应该会很有用的

在AndroidMenifest.xml中我们可以看到android:sharedUserId="android.uid.system"  
但是有了这句后，就无法对sd卡进行读写操作，比如在SD卡中创建一个新文件夹，是创建不成功的。但是如果把android:sharedUserId="android.uid.system"注释掉，就可以在SD卡进行IO操作了。