**[Android系统root破解原理分析](http://www.cnblogs.com/xiaoxiaoboke/archive/2012/02/13/2349719.html)**

     现在Android系统的root破解基本上成为大家的必备技能！网上也有很多中一键破解的软件，使root破解越来越容易。但是你思考过root破解的 原理吗？root破解的本质是什么呢？难道是利用了Linux kernal的漏洞吗？本文将简单对root的破解原理进行分析。

     网上有一篇文章已经对root破解的基本原理进行了简单介绍，大家可以先参考一下《[android root权限破解分析](http://hi.baidu.com/%CF%B1%B8%BE%C2%B1%C3%E6/blog/item/14c5dc296003ce225243c170.html" \t "_blank)》，本文只能说对root原理进行了方向性的描述，但是在一些具体的方面没有描述清楚。本文将会对其进行一些必要的扩展和补充。

     如果你进行过程序开发，在root过的手机上面获得root权限的代码如下：

1: Process process = Runtime.getRuntime().exec("su");

2: DataOutputStream os = new DataOutputStream(process.getOutputStream());

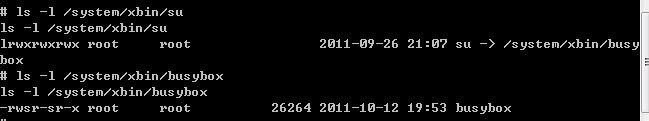
3: ......

4: os.writeBytes("exit\n");

5: os.flush();

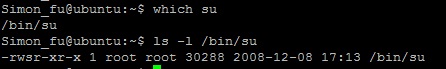
     从上面代码我们可以看到首先要运行su程序，其实root的秘密都在su程序中，《[android root权限破解分析](http://hi.baidu.com/%CF%B1%B8%BE%C2%B1%C3%E6/blog/item/14c5dc296003ce225243c170.html" \t "_blank)》中讲到Android系统默认的su程序只能root和shell可以用运行su，这个是安全的。如果把这个限制拿掉，就是root破解了！

     下面我们仔细分析一下程序是怎样获得root权限的，如果对Linux的su命令熟悉的朋友可能知道su程序都设置SUID位，我们查看一下我的手机（已经root破解）上的su权限设置，



      我们发现su的所有者和所有组都是root，是其实是busybox的软链接，我们查看busybox的属性发现，其设置了SUID和SGID，并且所有者和所有组都是root。SUID和SGID的作用是什么呢？如果你不太清楚，请参考《[Linux进程的实际用户ID和有效用户ID](http://my.unix-center.net/~Simon_fu/?p=607)》，这样运行busybox的普通用户，busybox运行过程中获得的是root的有效用户。su程序则是把自己启动一个新的程序，并把自己权限提升至root（我们前面提到su其实就是busybox，运行期它的权限是root，当然也有权限来提升自己的权限）。

     再强调一下不光root手机上su需要设置SUID，所有的Linux系统上的su程序都需要设置SUID位。请参考一下UC服务器的su的权限情况：



     我们发现su也设置了SUID位，这样普通用户也可以运行su程序，su程序会验证root密码，如果正确su程序可以把用户权限提高的root（因为其设置SUID位，运行期是root权限，这样其有权限提升自己的权限）。

     这样我们就可以看出其实Android系统的破解的根本原理就是替换掉系统中的su程序，因为系统中的默认su程序需要验证实际用户权限（只有root和 shell用户才有权运行系统默认的su程序，其他用户运行都会返回错误）。而破解后的su将不检查实际用户权限，这样普通的用户也将可以运行su程序， 也可以通过su程序将自己的权限提升。

     到这里大家对root破解不感到神秘了吧。root破解没有利用什么Linux内核漏洞（Linux内核不可能有这么大的漏洞存在），可以理解成root 破解就是在你系统中植入“木马su”，说它是“木马”一点儿都不为过，假如恶意程序在系统中运行也可以通过su来提升自己的权限的这样的结果将会是灾难性 的。所以一般情况下root过手机都会有一个SuperUser应用程序来让用户管理允许谁获得root权限，也算是给系统加了一层保险吧！

     如上是Simon的个人对root破解的一些认识，如果有错误的地方，欢迎朋友们指正。

[**Android系统root破解原理分析（续）**](http://www.cnblogs.com/xiaoxiaoboke/archive/2012/02/13/2349723.html)

上文《[Android系统root破解原理分析](http://my.unix-center.net/~Simon_fu/?p=1069)》介绍了Android系统root破解之后，应用程序获得root权限的原理。有一些网友提出对于root破解过程比较感兴趣，也提出了疑问。本文将会对这个root破解过程进行分析，来解答这个问题。

通过上文《[Android系统root破解原理分析](http://my.unix-center.net/~Simon_fu/?p=1069)》 的介绍大家应该明白了root破解过程的终极目标是替换掉系统中的su程序。但是要想替换掉系统中su程序本身就是需要root权限的，怎样在root破 解过程中获得root权限，成为我们研究的重点了。下面我们先清点一下我们需要破解系统情况，假设需要破解的Android系统具备如下条件：

1、可以通过adb连接到设备，一般意味着驱动程序已经安装。   
2、但是adb获得用户权限是shell用户，而不是root。

要想理解root破解过程我们首先需要了解一下adb工具，SDK中包含adb工具，设备端有adbd服务程序后台 运行，为开发机的adb程序提供服务，adbd的权限，决定了adb的权限。具体用户可查看/system/core/adb下的源码，查看 Android.mk你将会发现adb和adbd其实是一份代码，然后通过宏来编译。

查看adb.c的adb\_main函数你将会发现adbd中有如下代码：

1: int adb\_main(int is\_daemon)

2: {

3: ......

4: property\_get("ro.secure", value, "");

5: if (strcmp(value, "1") == 0) {

6: // don't run as root if ro.secure is set...

7: secure = 1;

8: ......

9: }

10:

11: if (secure) {

12: ......

13: setgid(AID\_SHELL);

14: setuid(AID\_SHELL);

15: ......

16: }

17: }

从中我们可以看到adbd会检测系统的ro.secure属性，如果该属性为1则将会把自己的用户权限降级成shell用户。一般设备出厂的时候在/default.prop文件中都会有：

1: ro.secure=1

这样将会使adbd启动的时候自动降级成shell用户。

然后我们再介绍一下adbd在什么时候启动的呢？答案是在init.rc中配置的系统服务，由init进程启动。我们查看init.rc中有如下内容：

1: # adbd is controlled by the persist.service.adb.enable system property

2: service adbd /sbin/adbd

3: disabled

对Android属性系统少有了解的朋友将会知道，在init.rc中配置的系统服务启动的时候都是root权限（因为init进行是root权限，其子程序也是root）。由此我们可以知道在adbd程序在执行：

1: /\* then switch user and group to "shell" \*/

2: setgid(AID\_SHELL);

3: setuid(AID\_SHELL);

代码之前都是root权限，只有执行这两句之后才变成shell权限的。

这样我们就可以引出root破解过程中获得root权限的方法了，那就是让以上面setgid和setuid函数执行失败，也就是降级失败，那就继续在root权限下面运行了。

这其实利用了一个RageAgainstTheCage漏洞，具体分析请参考《[Android adb setuid提权漏洞的分析](http://blog.claudxiao.net/2011/04/android-adb-setuid/" \t "_blank)》和《[RageAgainstTheCage](http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/)》。这里面做一个简单说明：

1、出厂设置的ro.secure属性为1，则adbd也将运行在shell用户权限下；  
  
2、adb工具创建的进程ratc也运行在shell用户权限下；

3、ratc一直创建子进程（ratc创建的子程序也 将会运行在shell用户权限下），紧接着子程序退出，形成僵尸进程，占用shell用户的进程资源，直到到达shell用户的进程数为 RLIMIT\_NPROC的时候（包括adbd、ratc及其子程序），这是ratc将会创建子进程失败。这时候杀掉adbd，adbd进程因为是 Android系统服务，将会被Android系统自动重启，这时候ratc也在竞争产生子程序。在adbd程序执行上面setgid和setuid之 前，ratc已经创建了一个新的子进程，那么shell用户的进程限额已经达到，则adbd进程执行setgid和setuid将会失败。根据代码我们发 现失败之后adbd将会继续执行。这样adbd进程将会运行在root权限下面了。  
  
3、这是重新用adb连接设备，则adb将会运行在root权限下面了。

通过上面的介绍我们发现利用RageAgainstTheCage漏洞，可以使adbd获得root权限，也就是adb获得了root权限。拿到root权限剩下的问题就好办了，复制破解之后的su程序到系统中（见上文《[Android系统root破解原理分析](http://my.unix-center.net/~Simon_fu/?p=1069)》的介绍），都是没有什么技术含量的事情了。

其实堵住adbd的这个漏洞其实也挺简单的：

1: /\* then switch user and group to "shell" \*/

2: if (setgid(AID\_SHELL) != 0) {

3: exit(1);

4: }

5: if (setuid(AID\_SHELL) != 0) {

6: exit(1);

7: }

如果发现setgid和setuid函数执行失败，则adbd进程异常退出，就把这个漏洞给堵上了。为什么这么多设 备都没有堵上这个漏洞呢？我觉得是设备厂商的策略（不排除傻X的厂商存在哦），虽然知道怎么封堵漏洞但是就是留着个后门给大家，让第三方给自己定制 rom，提高自己系统的易用性。

至此我们把root的过程和root之后系统情况都进行了介绍，相信你也不会对root破解再神秘了吧！你对本文和上文有什么意见欢迎和我探讨。

​

Android adb setuid提权漏洞的分析

[11条回复](http://www.claudxiao.net/2011/04/android-adb-setuid/#comments)

去年的Android adb setuid提权漏洞被用于各类root刷机，漏洞发现人[Sebastian Krahmer](http://c-skills.blogspot.com/" \t "_blank)公布的利用工具RageAgainstTheCage（rageagainstthecage-arm5.bin）被用于z4root等提权工具、Trojan.Android.Rootcager等恶意代码之中。下面我们来分析这一漏洞的产生原因。

The Android Exploid Crew小组在后来发布了一份PoC代码：[rageagainstthecage.c](http://blog.claudxiao.net/wp-content/uploads/2011/04/rageagainstthecage.c)。从这份代码开始着手。

在main(:72)函数中，首先获取了RLIMIT\_NPROC的值(:83)，这个值是Linux内核中定义的每个用户可以运行的最大进程数。

然后，调用find\_adb()函数(:94)来搜索Android系统中adb进程的PID，具体而言，该函数读取每个进程对应的文件的/proc/<pid>/cmdline，根据其是否等于”/sbin/adb”来判断是否adb进程。

接下来，fork了一个新的进程(:109)，父进程退出，而子进程继续。接下来，在113行创建一个管道。

|  |
| --- |
| rageagainstthecage.c |
| if (fork() > 0)      exit(0);    setsid();  pipe(pepe); |

重头戏发生在下面的122到138行，代码如下：

|  |
| --- |
| rageagainstthecage.c |
| if (fork() == 0) {      close(pepe[0]);      for (;;) {          if ((p = fork()) == 0) {              exit(0);          } else if (p < 0) {              if (new\_pids) {                  printf("\n[+] Forked %d childs.\n", pids);                  new\_pids = 0;                  write(pepe[1], &c, 1);                  close(pepe[1]);              }          } else {              ++pids;          }      }  } |

新建一个进程后，在子进程之中，exploit代码不断地fork()(:125)，而新的子进程不断退出，从而产生大量的僵尸进程（占据shell用户的进程数）。最终，进程数达到上限，fork()返回小于0，于是打印当前已经创建多少子进程，并向管道输入一个字符（:131）。

在这里，管道的作用是和(:122)fork出来的父进程同步，该进程在141行read这一管道，因而阻塞直至僵尸进程已经达到上限(:131)。

进一步的，exploit杀掉adb进程，并在系统检测到这一现象并重启一个adb之前，再一次fork()，将前一个adb留下的进程空位占据。最后，在152行，exploit调用wait\_for\_root\_adb()，等待系统重启一个adb，这个新建的adb就会具有root权限。

为什么在shell用户的进程数达到上限RLIMIT\_NPROC以后，新建的adb会具有root权限？我们来看adb的源码。

在<android\_src>/system/core/adb/adb.c的第918行，我们可以看到如下代码：

|  |
| --- |
| android\_src/system/core/adb/adb.c |
| /\* then switch user and group to "shell" \*/  if (setgid(AID\_SHELL) != 0) {      exit(1);  }  if (setuid(AID\_SHELL) != 0) {      exit(1);  } |

这已经是漏洞修补以后的代码。在漏洞最初被发现时，代码如下：

|  |
| --- |
| android\_src/system/core/adb/adb.c |
| /\* then switch user and group to "shell" \*/  setgid(AID\_SHELL);  setuid(AID\_SHELL); |

简而言之，原来没有检查setuid()函数的返回值。事实上，在此之前，adb.c中的代码都是以root权限运行，以完成部分初始化工作。在这一行，通过调用setuid()将用户从root切换回shell，但setuid()在shell用户进程数达到上限RLIMIT\_NPROC时，会失败，因此adb.c继续以root身份运行，而没有报错。

我们来看setuid()的man手册（man 2 setuid），其中有如下说明：

|  |
| --- |
| man 2 setuid |
| RETURN VALUE         On  success,  zero is returned.  On error, -1 is returned, and errno is         set appropriately.    ERRORS         EAGAIN The uid does not match the current uid and  uid  brings  process                over its RLIMIT\_NPROC resource limit. |

可以看到，setuid是可能发生错误的，并且在uid的进程数超过RLIMIT\_NPROC极限时，发生EAGAIN错误。

在android的源码中，setuid()定义于<android\_src>/bionic/libc/unistd/setuid.c，实际上引用了一个外部符号\_\_setuid，这个符号在<android\_src>/bionic/libc/arch\_xxx/syscalls/\_\_setuid.S中定义，最终是一个%eax=$\_\_NR\_setuid32，%ebx=uid的int 0×80中断。

因为只是要分析原理，我们不再鏖战于Android，转而看向Linux内核。

在最新的kernel2.6中，setuid()位于kernel/sys.c的682行，其中，在697行，一切正常的情况下，它会调用set\_user()来完成用户切换。

set\_user()实现于同一文件的587行，其中一部分代码如下：

|  |
| --- |
| kernel/sys.c |
| if (atomic\_read(&new\_user->processes) >= rlimit(RLIMIT\_NPROC) &&          new\_user != INIT\_USER) {      free\_uid(new\_user);      return -EAGAIN;  } |

含义很明显，当目标用户的进程数达到上限，那系统就不能再将一个进程分配给它，因而返回-EAGEIN。然后再setuid()中，直接跳过后面的代码，而返回错误。

至此，整个漏洞的原理已经分析完毕。整理如下：

1、在Android的shell用户下，制造大量的僵尸进程，直至达到shell用户的进程数上限RLIMIT\_NPROC；

2、kill当前系统中的adb进程，并再次占据其进程位置以保持达到上限；

3、系统会在一段时间后重启一个adb进程，该进程最初是root用户，在完成少许初始化工作后，调用setuid()切换至shell用户；

4、此时shell用户的进程数已经达到上限，所以setuid()失败，返回-1，并且用户更换没有完成，adb还是root权限；

5、adb没有检查setuid()的返回值，继续后续的工作，因此产生了一个具有root权限的adb进程，可以被用于与用户的下一步交互。

实际上，setuid在目标用户进程数达到RLIMIT\_NPROC极限时返回错误，这一问题可能产生的安全隐患最早可以追溯到[2000年](http://www.mail-archive.com/freebsd-hackers@freebsd.org/msg08626.html" \t "_blank)。而在2006年，出现了真正利用这一编码问题的漏洞（[CVE-2006-2607](http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-2607" \t "_blank)）。

因此，这并不是一个全新的漏洞。我们可以得出几点结论：

1、函数返回值一直是忽略的对象，因为getuid()永远不会失败，程序员可能会认为setuid()也不会失败——至少没有遇到过，因此忽略了对返回值的检查。检查一个系统函数是否调用失败是一个常识，但又是很麻烦的事，如果为了省事而忽略，问题就可能产生了。

2、Android下的安全问题，很多并非全新的，而且个人判断将来还会有大量漏洞、恶意代码产生于传统思路，而作用于新的平台。面对这一新的平台，我们是否能抢先于攻击者做好防范准备，是一个需要我们思考和实践的问题。