Abstract

The security model of the Android OS is based on the effective combination of a number of well-known security mechanisms (e.g. statically defined permissions for applications, the isolation offered by the Dalvik Virtual Machine, and the well-known Linux discretionary access control model). Although each security mechanism has been extensively tested and proved to be effective in isolation, their combination may suffer from unexpected security flaws. We show that this is actually the case by presenting a severe vulnerability in Android related to the application launching flow. This vulnerability is based on a security flaw affecting a kernel-level socket (namely, the Zygote socket). We also present an exploit of the vulnerability that allows a malicious application to mount a severe Denial-of-Service attack that makes the Android devices become totally unresponsive. Besides explaining the vulnerability (which affects all versions of Android up to version 4.0.3) we propose two fixes. One of the two fixes has been adopted in the official release of Android, starting with version 4.1. We empirically assess the impact of the vulnerability as well as the efficacy of the countermeasures on the end user. We conclude by extending our security analysis to the whole set of sockets, showing that other sockets do not suffer from the same vulnerability as the Zygote one.

1Corresponding author
1. Introduction

By leveraging a generic Linux kernel, the Android OS is built out of a layered architecture that runs on a wide variety of devices and supports the execution of a large number of applications available for download both inside and outside the Google Play store. Since most applications are developed by third-parties, security is a major concern. The Android security model tackles the problem by striving to attain the following design goal:

A central design point of the Android security architecture is that no application, by default, has permission to perform any operation that would adversely impact other applications, the operating system, or the user.


This goal is pursued through a number of cross-layer security mechanisms aimed at isolating applications from each other. These mechanisms are built out of basic security mechanisms available in the individual layers of the Android stack. For instance, application sandboxing\(^2\) is achieved by combining the isolation guaranteed by the use of Dalvik Virtual Machines together with the native Discretionary Access Control (DAC) offered by Linux and with a set of statically defined Android permissions specifying which operations each application is allowed to execute. These permissions regulate access to hardware components such as the telephony subsystem as well as accesses to sensitive data (e.g. messaging history) and to inter-process communication mechanisms (e.g. intent and broadcast). Each Android application comes up with a set of permissions (i.e. corresponding to operations the application requires to execute during its life cycle) that the user must explicitly grant during installation or upgrade. According to their potential risk, Android

---

\(^{2}\)http://developer.android.com/guide/practices/security.html

*This work has been partially founded by EU project FP7-257876 SPaClIoS.*
permissions are divided into four categories, namely (from lower to higher risk) normal, dangerous, signature and signatureOrSystem\(^3\).

The individual mechanisms are generally well-known and have been thoroughly tested i.e. the isolation offered by virtual machines has been tested for Java applets, access control for Unix/Linux is well-known, and explicit permissions have been tested inside the Java security architecture. However, this is not the case for the specific combination deployed in Android: the cross-layer interaction among the available mechanisms has not been fully explored yet and may therefore suffer from security weaknesses.

In this paper we present a serious vulnerability in the Android Launching Flow. The Android Launching Flow is a sequence of activities that are normally carried out by the system whenever a new application is launched. In Android, applications cannot directly fork new processes and a single process, the Zygote Process, is permitted to carry out this crucial activity. Requests for the creation of new processes are therefore sent to the Zygote Process through a specific socket, called the Zygote Socket.

Up to OS version 4.0.3 included, the Android Launching Flow suffered from a vulnerability that allows a malicious application to side-step all Android security checks and force the system to fork an unbounded number of processes thereby making the device completely unresponsive. In order to exploit the vulnerability, a malicious application does not need any Android permission.

Rebooting the device does not necessarily help as a malicious application can be crafted in such a way to be automatically launched at boot-time without asking for the user’s explicit approval of any permission upon installation. This makes the malicious applications particularly mischievous as even the most cautious user may consider it as harmless.

More in general, the existence of this vulnerability shows that the aforementioned fundamental design goal of the Android Security Framework, i.e. the impossibility for an application to adversely impact another application, is not met. Indeed, a malicious application can severely affect all other applications, the operating system, and ultimately the user’s experience.

In the paper we also propose two solutions to the problem: the first solution involves minimal changes in the Android security model, the second requires a number of additional cross-layer checks. We show that both solu-

\(^3\)http://developer.android.com/guide/topics/manifest/permission-element.html
tions are equally effective in fixing the vulnerability.
In the paper we also report on experimental results confirming that:

- all versions of Android OS up to 4.0.3 included (that means 80% of Android devices in March 2013) suffer from the vulnerability,
- our proposed fixes effectively counter the DoS attack, and
- the two most recent versions of Android (namely, versions 4.1 and 4.2) no longer suffer from the vulnerability.

We have promptly informed of our findings both the Android Security Team and the US-CERT. In response to our findings, the Android OS has been swiftly patched by incorporating (a simplified variant of) one of our two fixes in the two most recent versions of Android. Moreover, the US-CERT has recently issued a vulnerability note (CVE-2011-3918) that describes the issue and evaluates its severity as HIGH.\footnote{http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3918}

This paper revises and extends our previous work (Armando et al., 2012b) in a number of ways:

- we provide a more detailed analysis of the Android architecture and of its security mechanisms;
- we provide an extended description of the vulnerability as well as of the countermeasures, by adding details that are necessary to fully understand and reproduce the problem, and the proposed fixes;
- we extend the experiments related to both the vulnerability and the countermeasures by involving end users;
- we provide a new section describing the recent developments in the Android architecture related to the discussed vulnerability and the patch adopted in the latest versions of Android;
- we assess the security of two additional sockets used by Android that are very similar to the Zygote socket.
- we report the results of an experimental analysis that confirms the effectiveness of the fixes in countering the DoS attack and that the patches do not affect the nominal behavior of the system.

\footnote{http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3918}
Structure of the paper. In the next section we provide a brief description of the Android architecture. In Section 3 we describe the security mechanisms used in Android. In Section 4 we present the vulnerability and in Section 5 we illustrate two possible solutions. In Section 6 we present our experimental results. In Section 7 we discuss the effectiveness of the fix incorporated in Android and extend our security analysis to other parts of Android. In Section 8 we compare our work with the current Android literature and we conclude in Section 9 with some final remarks.

2. The Android Architecture

The Android Architecture consists of 5 layers. The Linux kernel lives in the bottom layer (henceforth the Linux layer). The remaining four layers are Android-specific and we therefore collectively call them Android layers.

2.1. The Android Layers

The Android Layers are (from top to bottom): the Application layer, the Application Framework layer, the Android Runtime layer, and the Libraries layer:

- **Application Layer.** Applications are at the top of the stack and comprise both user and system applications that have been installed and executed on the device. They are made by a set of components each performing a different role in the logic of the application\(^5\). Each application comes with a set of permissions on the OS that must be granted by the user upon installation in order to allow the application to execute properly. The complete list of the Android permissions can be found here\(^6\).

- **Application Framework Layer.** The Application Framework provides the main services of the platform that are exposed to applications as a set of APIs. This layer provides the System Server, that is a process containing the main modules for managing the device (e.g. the Activity Manager and the Package Manager) and for interacting with the

\(^{5}\)See http://developer.android.com/guide/components/fundamentals.html for more information on Android applications and related components.

\(^{6}\)http://developer.android.com/guide/topics/security/permissions.html
underlying Linux drivers (e.g. the Telephony Manager and the Location Manager that handle the telephony service and the GPS module, respectively).

- **Android Runtime Layer.** This layer consists of the Dalvik Virtual Machine (Dalvik VM, for short), i.e. the Android runtime core component that executes application files built in the Dalvik Executable format (.dex). The Dalvik VM is specifically optimized for efficient concurrent execution of virtual machines in a resource constrained environment.

- **Libraries Layer.** The Libraries layer contains a set of C/C++ libraries that support the invocation of basic kernel functionalities. They are widely used by Application Framework services to invoke protected Linux operations and to access data stored on the device. Examples of libraries are the bionic libc (a customized implementation of libc for Android) and SQLite.

### 2.2. The Linux Layer

Android relies on the Linux kernel version 2.6 or later for core system functionalities. These functionalities include the access to peripherals and Inter-Process Communication (IPC). Device peripherals (e.g. GPS antenna, Bluetooth/Wireless/3G modules, camera, accelerometer) are accessed through Linux drivers. Each peripheral has a driver installed as a kernel module. Whenever an upper layer requires (an authorized) access to a peripheral, the kernel queries the corresponding module in kernel mode (i.e. with maximum privileges).

Communication between applications in Android is carried out through the Binder driver and native Unix Domain Sockets.

- The Binder IPC is the primary communication mechanism in Android. Each process registers itself to the Binder driver and gets back a file descriptor. A process aiming to communicate with another process can send data through it. The Binder driver receives the data and sends them directly to the destination process.

- Unix Domain Sockets (hereafter, sockets) are kernel files that Android specifically uses to receive commands for system operations. Each
socket is controlled by a process. Services from the Application Framework write commands on a socket to require the execution of the corresponding kernel operation. The controlling process invokes kernel operations by accessing specific procedures at the Libraries layer.

Table 1: Sockets used in Android

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbus</td>
<td>used to support communication between the Bluetooth library (Bluez) and the System Server</td>
</tr>
<tr>
<td>installld</td>
<td>used to send commands to the Linux installer daemon which is responsible for retrieving and installing applications</td>
</tr>
<tr>
<td>dnsproxyd</td>
<td>handles communications with the Linux DNS proxy daemon</td>
</tr>
<tr>
<td>keystore</td>
<td>used to store application certificates and cryptographic keys</td>
</tr>
<tr>
<td>netd</td>
<td>provides access the network via the Linux net daemon</td>
</tr>
<tr>
<td>property_service</td>
<td>used to store system-wide properties</td>
</tr>
<tr>
<td>rild</td>
<td>used to control the Linux RILD daemon, which handles telephone/3G/GPRS modules</td>
</tr>
<tr>
<td>rild-debug</td>
<td>used to debug the rild socket</td>
</tr>
<tr>
<td>void</td>
<td>used to send messages to void daemon, which detects, mounts and monitors the status of the SD card</td>
</tr>
<tr>
<td>zygote</td>
<td>used to build new Linux processes hosting newly launched applications</td>
</tr>
</tbody>
</table>

The sockets used by Android are described in Table 1. Since sockets trigger protected system operations, they are normally used by trusted services at the Application Framework layer. However, although their use by the Application layer is discouraged, applications can directly use them.

2.3. Cross-layer Interactions in Android

In Android, the interactions among layers is carried out through appropriate calls:  

---

1. **Binder calls.** Binder calls allow to invoke the Binder driver and carry out an IPC communication. A Binder call can be invoked by any Android layer and allows to delivery a message by means of serialized Java objects to any other Android layer.

2. **JNI calls.** Java Native Interface (JNI) calls are invoked by Java layers (i.e. Application, Application Framework, and Application Runtime) to get access to the C/C++ methods at Libraries layer.

3. **Socket calls.** Requests for services at the Linux layer issued through sockets. As stated above, such calls are used by trusted services to request kernel functionalities.

4. **System calls.** System calls are used to invoke a native functionality in kernel mode. They are directly invoked from Libraries layer in response to a request coming from upper layers.

5. **Function calls.** Generic, intra-component calls that can be invoked within each layer.

6. **Dynamic load calls.** Dynamic load calls allow Application and Application Framework layers to dynamically include and access functionalities available in the pre-compiled libraries of Libraries layer.

By combining different sets of calls, the Android Stack provides all the needed functionalities to the applications. We informally define a (partially ordered) set of calls aimed at accomplishing a given functionality as a flow. In the following, we investigate the flow related to the launch of an Android application.

### 2.4. The Application Launching Flow

In Android new applications can be launched by tapping the corresponding icon on the home screen. Tapping forces the activity component of the home screen application to start a flow (depicted in Fig. 1) which involves four entities:

- an *Activity component* (i.e. an application component corresponding to an active interface on the screen) from which the user can require the launch of a new application (e.g. the *home* screen); due to sandboxing, the application is mapped on a generic non-privileged user at the Linux layer;

---

8[^note]

• the Activity Manager Service at the Application Framework layer; it is contained in the System Server, which is mapped on a generic non-privileged user at the Linux layer;

• a Zygote process, owned by root, containing a Dalvik VM (at the Runtime layer), and a set of functions, i.e. the Zygote Library (at the Libraries layer);

• a Zygote socket at the Linux layer, owned by the root user.

The Activity Manager Service is the trusted service at the Application Framework layer in charge of launching new applications. It is activated in response to a request from an Activity component and issues appropriate
commands to the Zygote socket. The Zygote process, created during the Linux boot-strap, is the controlling process of the Zygote socket. The Zygote process is able to invoke fork system calls at the Linux layer, in order to build a new process to host the launching application. In detail, the Zygote process listens for incoming commands on the Zygote socket and generates a new process by forking itself. Differently from what happens in a normal Linux system, specialization of the child process is not obtained by loading a new executable image, but only by loading the Java classes of the specific application inside the Dalvik VM hosted in the child process.

When an application is launched a `startActivity` call is sent to the Activity Manager Service by means of a Binder call. The Activity Manager Service determines if the application has enough permissions to launch a new application. If so, it calls the method `startActivityLocked` which checks whether the launching application has already an attached process at the Linux layer or if a new one is needed. The first case happens when an instance of the application has been previously started and it is currently executing in background; when this is the case, the Activity Manager Service gets the corresponding process and brings back the application to foreground. Otherwise, the Activity Manager Service calls `Process.start()`, a method of the static class `android.os.Process`. This method connects the Activity Manager Service to the the Zygote socket through a socket call, sending parameters for the fork command.

The command sent to the Zygote socket includes the name of the Java class whose static `main` method will be invoked to specialize the child process. The Activity Manager Server uses a standard class (i.e. `android.app.-ActivityThread`) when forking. In this class, a binding operation between the Linux process and an Android application is explicitly attempted. The `main` method of `ActivityThread` checks whether the application is contained in an appropriate list stored in the Activity Manager Service containing the installed applications that are not currently running. If the application being launched is contained in the list, the `main` method loads the corresponding code in the new process. Otherwise, i.e. if no application is available for binding, the same class asks the Linux kernel to kill the process. If the spawning of the new process and the binding operation succeed, the Zygote process returns its child’s PID to the Activity Manager Service and the flow successfully terminates.
3. Android Security Mechanisms

Each layer in the Android stack (except the Libraries layer) comes with its own security mechanisms:

- **Application layer (Android Permissions)**. Each application comes with a file named `AndroidManifest.xml` that contains the permissions that the application may require during execution. During installation the user is asked to grant all the permissions specified in the manifest.

- **Application Framework (Permission Enforcement)**. Services at this layer enforces the permissions specified in the manifest and granted by the user during installation.

- **Runtime (VM Isolation)**. Each application is executed in a separate Dalvik VM machine. This ensures isolation among applications.

- **Linux (Access Control)**. As in any Linux Kernel, resources are mapped into files (e.g. sockets, drivers). The Linux Discretionary Access Control (DAC) model associates each file with an owner and a group. Then, DAC model allows the owner to assign an access control list (i.e. read, write, and/or execute) on each file to the owner itself (UID), the owner’s group (GID) and other users.

Android achieves sandboxing of the applications by binding each Android application to a separate Linux user, thereby combining the memory separation due to the execution of applications inside different Dalvik VMs with the resource access isolation provided by Linux access control.

Once an application is installed on the device (i.e. the user accepts all required permissions) a new user at the Linux layer is created and the corresponding user id (UID) is bound to the installed application. As stated in the previous section, once the application is launched, a new process, with such UID, and a novel Dalvik VM are created in order to execute the application.

This solution forces any non privileged UID to have at most one process running (i.e. the one containing the running application). Rarely, more than one active process for the same UID can be allowed if explicitly requested in the `AndroidManifest.xml`. However, the maximum number of active processes is upper-bounded by the number of components composing the application.

As mentioned above, the Zygote socket is owned by root and it is controlled by a root process (i.e. the Zygote process). However, the Zygote socket has permissions 666 (i.e. rw-rw-rw). This implies that any Android application can read and write it and hence send commands to the Zygote process. This choice is justified by the need of the System Server process (whose UID is 1000, it belongs to a non-privileged Linux user, and it does not belong to the root group) to request the spawning of new processes. The message sent through the socket follows a specific format, listed as follow:

```
1 "--runtime-init --setuid=<uid> --setgid=<gid>
2 [--enable-safemode] [--enable-debugger]
3 [--enable-checkjni] [--enable-assert]
4 --setgroups= <group list>
5 --capabilities= <capabilities list>
6 --nice-name=<niceName>
7 <ClassName>
```

The fork request for a new process includes the possibility of setting:

1. a specific UID and GID (cf. line 1),
2. optional parameters, used to enable safemode (which disables the Java Just-In-Time compiler) or to permit global debugging (enable-debugger) (cf. line 2),
3. optional parameters, used to enable JNI code debugging (enable-checkjni) and assertions (used to test assumptions about a program) (cf. line 3),
4. a sets of supplementary group IDs, which can be used to allow a process to access specific Linux resources (i.e. sdcard_rw GID allows a process to access SD-card files)(cf. line 4),
5. additional capabilities (tokens for specific Linux privileges, normally assigned only to root user—cf. line 5),
6. a human-readable name for the process list (cf. line 6),
7. a Java class that will be used to specialize the new process(cf. line 7).

To avoid misuse, the Zygote process enforces a security policy that restricts the execution of the commands received on the Zygote socket. This policy is implemented in the runOnce() function and enforced on the command parameters:
boolean runOnce()
throws ZygoteInit.MethodAndArgsCaller{
    ...
    applyUidSecurityPolicy(parsedArgs, peer)
    applyCapabilitiesSecurityPolicy(parsedArgs, peer);
    applyDebuggerSecurityPolicy(parsedArgs);
    applyRlimitSecurityPolicy(parsedArgs, peer);
    ...
}

The security policy prevents from

1. issuing the command that explicitly specifies a UID and a GID for the
   creation of new processes if the requester is not root or the System
   Server (cf. line 4),
2. creating a child process with more capabilities than its parent (cf. line 5),
   and
3. enabling debug-mode flags and specifying rlimit bound if the requester
   is not root or the System is not in “factory test mode” (cf. lines 6 and
   7).

However, only a few checks are carried out on the (static) class used to
customize the child process, namely (i) whether the class contains a static
main() method and (ii) whether it belongs to the System package, which is
the only one accepted by the Dalvik System Class loader.

4. A Vulnerability in the Application Launching Flow

   The security checks performed in the class used to customize the child
   process do not include a proper check on the identity (UID) of the requesting
   process. This allows any Linux process (and its bound Android application
   or service) to send malicious fork commands to the Zygote socket by using
   the following arguments:
   
   1. --runtime-init --nice-name=dummy
   2. com.android.internal.util.WithFramework

---

13
Notice that this fork request does not include any information on the Linux IDs (UID, GID) or capabilities of the requesting process, thereby bypassing both the `applyUidSecurityPolicy` and the `applyCapabilitiesSecurityPolicy` checks. Moreover, since no optional parameter is provided, the `applyDebuggerSecurityPolicy` policy and the `applyRlimitSecurityPolicy` policy are trivially satisfied (as long as a valid static class is provided).

We discovered that by using the System static class `com.android.-internal.util.WithFramework` it is possible to force the Zygote process to fork a dummy process which is kept alive at the Linux layer. Such a class does not perform any binding operation between the new process and an Android application and therefore it does not trigger the removal of unbound new processes as the default `android.app.ActivityThread` class does. This leads to a persistent process which occupies memory resources in the device. By flooding the Zygote socket with such requests, an increasingly large number of dummy processes is created until the memory of the device is exhausted, successfully executing a fork bomb attack. The security mechanisms in the Android layers are unable to notice the generation of dummy processes and, consequently, to intervene.

It is worth pointing out that the creation of processes at the Linux layer is legal and managed directly by the kernel, thereby not violating any rule of the Linux DAC model. Besides, dummy processes are owned by root, thus also the sandboxing rules (i.e. a single process for each unprivileged user) are ineffective. Therefore, none of the involved security mechanisms (both intra and cross-layer) is able to recognize such behavior as malicious.

As soon as the dummy processes consume all the available resources, a safety mechanism reboots the device. Thus, by launching the attack during boot-strapping, it is possible to lock the device into an endless boot-loop, thereby denying the use of the device. Notice also that to mount the attack, the malicious application does not require any permission (i.e. no permission must be accepted by the user during installation) and therefore it looks harmless to the user upon installation.

5. Countermeasures

We have identified two possible approaches to fix the previously described vulnerability:

1. **Zygote process fix.** This fix consists of checking whether the fork request to the Zygote process comes from a legal source (at present,
only the System Server, although this patch can be easily adapted to future developments).

2. **Zygote socket fix.** This fix restricts the permissions on the Zygote socket at the Linux layer.

5.1. *Checking Fork Requests inside the Zygote Process*

As remarked in Section 4, the Zygote process does not perform any check on the identity of the process requesting the fork operation. Nevertheless, the Zygote socket, created during the boot-strap of the Linux system, has a *credential passing mechanism* which allows to identify endpoints connected to the socket by means of their PID (i.e. Process ID), UID and GID.

This implies that the Zygote process can retrieve the identity (i.e. UID) of the requesting process. The extended policy takes advantage of this feature and applies a new filter based on the UID of the requesting process. In particular, since the System Server process has UID 1000 (statically defined), the extended security policy filters the requests reaching the Zygote socket by accepting only fork requests from UID 1000 and UID 0 (root):

```java
void applyForkSecurityPolicy(peer){
    int peerUid = peer.getUid();
    if (peerUid == 0 || peerUid == Process.SYSTEM_UID){
        // Root or System can send commands
        Log.d("ZYGOTE_PATCH", "root or SS request: OK");
    }
    else{
        Log.d("ZYGOTE_PATCH", "user request "+ peerUid + " blocked");
        throw new ZygoteSecurityException("user UID"+peerUid +" tries to fork new process");
    }
}
```

We implemented the previous policy by adding this check at the end of the native policy in the `runOnce()` method of the Zygote process.

5.2. *Modifying the Linux Permissions of the Zygote socket*

The idea is to reduce the Linux permissions for the Zygote socket. Currently, the Zygote socket is owned by root and the associated permissions
are 666 (i.e. rw-rw-rw). It is possible to modify both the owner (no root) and permissions of Zygote socket from 666 to 660 (i.e. rw-rw--). In this way, the System Server retains read and write access. We implemented this modification in three steps:

1. We created a new owner for the Zygote socket. To this end we added a new UID (namely 9988) in the file `android_filesystem_config.h` which contains statically assigned UIDs for Android system services. Then, in the same file we associated the new UID to an ad hoc user `zygote_socket`.

```c
#define AID_ZYGSOCKET 9988 /* Zygote socket UID;*/
#define AID_MISC 9998 /* access to misc storage*/
#define AID_NOBODY 9999
#define AID_APP 10000 /* first user app*/
```

```c
static struct android_id_info android_ids[] = {
    { "root", AID_ROOT, },
    { "system", AID_SYSTEM, },
    { "misc", AID_MISC, },
    { "zygote_socket", AID_ZYGSOCKET, },
    { "nobody", AID_NOBODY, },
};
```

2. We changed the owner and the permissions of the Zygote socket. We associated the user `zygote_socket` with the Zygote socket by modifying `init.rc` and by setting its permissions to 660.

```bash
service zygote /system/bin/app_process --Xzygote /
    system/bin --zygote --start-system-server
socket zygote stream 660 zygote_socket zygote_socket
    onrestart write /sys/android_power/
    request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```
3. We included the UID of the Zygote socket owner in the group list of the System Server. Since the System Server is also generated through a fork request to the Zygote process, we modified the parameter of the fork command corresponding to the set of groups the new process belongs to. We added the UID to such set as follows:

```java
String args[] = {
    "--setuid=1000",
    "--setgid=1000",
    "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003,9988",
    "--capabilities=130104352,130104352",
    "--runtime-init",
    "--nice-name=system_server",
    "com.android.server.SystemServer",
};
```

This solution has the additional advantage that it creates a restricted group that may be used in future developments without requiring further system changes.

6. Experimental Results

We tested the vulnerability on all versions of Android OS currently available (i.e. up to version 4.1). To this end we developed DoSChecker, a malicious application implementing the steps described in Section 4. The testing activity was carried out using both actual and simulated devices. All our tests led to the generation of an unbounded number of dummy processes, thus revealing that all versions considered suffer from the vulnerability described in this paper.

The Testing Environment. We used a laptop equipped with Android SDK r16. We tested the actual devices listed in Table 2 using the Android versions indicated in the same table. Testing of the versions that do not occur in Table 2 (e.g. version 4.1 and version 2.1), has been carried out using an Android device emulator. The behavior of the actual and simulated devices has been traced with Adb Logcat tool and Adb shell via a Windows shell.
Since the execution of a DoS attack depends on the amount of resources, we used actual devices (listed in Table 2) with heterogeneous hardware in order to assess the relation between the availability of resources and the time spent to accomplish a successful attack.

6.1. Exploiting the Vulnerability

*Testing with Actual Devices.* Once the DoSChecker application has been activated on a device, devices with limited resources (e.g. LG Optimus One) freeze in less than a minute while others freeze in at most 2 minutes (e.g. Galaxy Tab). Our tests show an almost linear dependence of the duration of the attack on the amount of resources available in the device. During the attack, the user experiences a progressive reduction of the system responsiveness that ends with the system crash and reboot. While the number of dummy processes increases, Android tries to kill legitimate applications to free resources, but it has no access to the dummy processes created by DoSChecker. This behavior ultimately leads to the killing of other application processes including system processes (such as the home process). In several cases the *System Server* crashes too. Once an application is killed, Android tries to restart it after a variable period of time, but DoSChecker fills the memory just freed with dummy processes, causing the failure of the restart procedure. Once physical resources are exhausted and Android is not able to restart applications the device is rebooted. DoSChecker has the same behavior both on standard and rooted (i.e. devices where non-system software components may temporary acquire root permissions) Android devices.

*Testing with Emulated Devices.* The use of the Android emulator allowed us to check the DoS vulnerability on Android versions such as Android 2.1 and 3.2 for which we do not own physical devices. Also in this case the experiments lead to the forking of an unbounded number of processes. However, we observed that in the emulated environment, where the amount of available resources depends on the host PC, the number of dummy processes generated would surely overcome the hardware capability of any device currently available on the market.

*Running DoSChecker as a Boot Service.* Since the exploitation of the vulnerability eventually leads to the reboot of the device, we added DoSChecker as a service starting at boot. This can be readily done by adding a Java class in DoSChecker that, acting as a Broadcast Receiver, intercepts the system
Table 2: Devices used in the experiments

<table>
<thead>
<tr>
<th>Device Model</th>
<th>RAM</th>
<th>Android Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lg Optimus One p550</td>
<td>512 MB</td>
<td>2.2.3, 2.2.4 (stock LG), 2.2.4 (rooted)</td>
</tr>
<tr>
<td>Lg Optimus 3D</td>
<td>512 MB</td>
<td>2.2.3, 2.2.4 (stock LG), 2.2.4 (rooted)</td>
</tr>
<tr>
<td>Lg Optimus L3</td>
<td>384 MB</td>
<td>2.3 (with Optimus UI)</td>
</tr>
<tr>
<td>Samsung Galaxy S</td>
<td>512 MB</td>
<td>2.3.4 (stock Samsung), 2.3.7 (Cyanogen 7)</td>
</tr>
<tr>
<td>Samsung Next GT-S5570</td>
<td>279 MB</td>
<td>2.3.4 (stock Samsung), 2.3.4 (rooted)</td>
</tr>
<tr>
<td>Samsung Galaxy Tab 7.1</td>
<td>1 GB</td>
<td>3.1 (stock Samsung), 3.1 (rooted)</td>
</tr>
<tr>
<td>Samsung Galaxy Nexus</td>
<td>1 GB</td>
<td>4.0 (stock and rooted), 4.0.3 (stock)</td>
</tr>
<tr>
<td>HTC Desire HD</td>
<td>768 MB</td>
<td>2.3.2 (stock HTC), 2.3.2 (rooted)</td>
</tr>
<tr>
<td>HTC Sensation XL</td>
<td>768 MB</td>
<td>2.3 (stock), 4.0 (rooted)</td>
</tr>
<tr>
<td>Motorola Droid RAZR MAXX</td>
<td>1 GB</td>
<td>2.3.3 (stock Motorola)</td>
</tr>
</tbody>
</table>
broadcast messages and starts DoSChecker as a service whenever a bootstrap terminates (i.e. when the system broadcast message android.intent.action. BOOT_COMPLETED is received).

Interestingly, due to an open bug\(^9\), the permission (i.e. android.permission .RECEIVE_BOOT_COMPLETED) is not actually required to launch an application at boot time in Android versions previous than 4.0.3. Furthermore, since the protection level of such permission is set to normal, the system automatically grants this type of permission to a requesting application at installation, without asking for the user’s explicit approval\(^{10}\). To sum up, at install time the user is unable to notice that DoSChecker may launch at boot time even in the newer versions of Android (i.e. 4.0.3 and above), unless he explicitly revises the list of permissions of the application.

Our tests show that the exploitation of the vulnerability at boot prevents Android from successfully completing the reboot process, thus freezing the device. In order to recover the usability of the device it is necessary either to identify and manually uninstall the malicious application via the Android Debug Bridge\(^{11}\) (also known as the adb tool) or to reinstall a fresh Android distribution (such procedure is also called flashing).

The User Experience. In order to assess the impact of the vulnerability on the end user, we embedded the DoSChecker code in a service component and in a Broadcast Receiver component of a customized weather application. Once the user executes the weather application, the DoSChecker service is activated and waits for a random period of time before executing the exploit code.

We selected a group of 10 users owning Android devices in our university. We asked them to install the weather application as well as other 5 harmless applications. Users reported that during the normal usage the device started being less responsive and it became totally unresponsive in few minutes at most. In some cases, the device rebooted endlessly. Neither the Android Task Manager nor other similar utilities like Advanced Task Killer\(^{12}\) were useful, since none of the users was able to relate the unexpected behavior of the

\(^9\)http://code.google.com/p/android/issues/detail?id=14044
\(^{10}\)http://developer.android.com/reference/android/R.styleable.html#Android-ManifestPermission_protectionLevel
\(^{11}\)http://developer.android.com/tools/help/adb.html
device with the weather application and, consequently, to stop or uninstall it during the vulnerability exploitation. Furthermore, when the device has become fully unusable, no application can be launched. All testing devices have been manually restored via *adb* shell and the a-priori knowledge about the cause of the problem. A user with no such knowledge, however, can only adopt a trial and error approach.

### 6.2. Effectiveness of the Fixes

To assess the effectiveness of the fixes described in Section 5 we built two patched variants of each available version of Android. For each variant we thus generated two images called `system.img` and `ramdisk.img`. The former contains all the (recompiled) system classes, the latter corresponds to the Android RAM disk image that is loaded during boot-strapping and that includes, among others, the `init.rc` file. The Zygote process fix, which extends the security policy applied by the Zygote process, affects `system.img` only. The Zygote socket fix, which also requires a modification of the `init.rc` file, affects both `system.img` and `ramdisk.img`.

We then defined the following experimental set up with the twofold aim of verifying the effectiveness of the fixes in countering the DoS attack and of verifying that the patches do not affect the nominal behavior of the system. We selected a population of 20 users (different from those involved in the vulnerability tests) with varying expertise (e.g. students, professors, technicians, and clerks) and randomly assigned each user with a device among those in Table 2. On each device we installed:

1. a patched Android version from our builds,
2. the *top 30 free applications* currently available on *Google Play*,
3. our malicious weather application containing the DoS Checker code.

The device has been configured in such a way to start the background execution of the malicious weather application during bootstrapping and to randomly activate the malicious code ten times in a week.

Then we delivered the device to each user for a week, asking him to use it and report performance degradation or malfunctioning. We provided each user with three different device/OS versions during the testing phase. In this way, we were able to have each patched Android version tested on multiple

---

physical devices by different users. No user experienced any unexpected behavior or performance degradation during the experiment.

7. Recent developments

In this section, we discuss recent developments related to the Zygote vulnerability and the adoption of the fixes. Moreover, we provide a security assessment of two additional sockets that are used by Android to carry out security critical operations.

7.1. Fixing the Android OS

We have promptly reported our findings to Google, Android and the US-CERT. The vulnerability has been registered in the CVE database and has been assigned identifier CVE-2011-3918\textsuperscript{14}. The US National Vulnerability Database has assigned a high degree of severity (CVSS score 7.2/10) to the vulnerability and the ease of exploitation has been assigned the maximum score (10/10)\textsuperscript{15}.

We proposed our two fixes to the Android Security Team and we jointly worked on a single final patch (named Zygote patch). The Zygote patch is a slight customization of the Zygote socket fix described in Section 5: the Zygote socket fix has been modified by changing the GID of the Zygote socket from root to system (corresponding to the group of services at the Application Framework layer), thereby avoiding the adoption of an ad-hoc Linux user (i.e. zygote_socket) for the Zygote socket. The Zygote patch has been tested using the DoSChecker application on both actual devices (listed in Table 1) and emulated machines. The new tests executions show that, comparing to our original patches, the new one is identically effective.

The Android Security Team has adopted the Zygote patch in version 4.1 and 4.2. Furthermore, a fixing upgrade has been released for version 4.0.3. Moreover, custom ROMs developed by third-parties (e.g. Cyanogen-Mod, Android Open Kang Project (AOKP)) have introduced our original Zygote Socket fix.

Official patches are delivered directly to devices via OTA (Over-the-air) upgrades. Users with older Android versions (i.e. lower than v. 4.0.3) are

\textsuperscript{14}http://cve.mitre.org/cgi-bin/cvename.cgi?name=2011-3918
\textsuperscript{15}http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3918
strongly recommended to upgrade their devices to newer versions once released by the device vendor. Alternatively, end users may get the source code of Android and apply one of the solution proposed in Sect. 5 manually. Such modifications require Android to be compiled for the specific hardware of the mobile device and finally flashed on the device. This solution can result burdensome for average users which are not confident with Java/C programming and Android operating system fundamentals.

7.2. Attack Exploitability

According to official statistics\textsuperscript{16}, the 57.4\% of in-use Android devices have no official fixing upgrade (i.e. distribution lower than 4.0.3 version) while the 28.6\% (those equipped with 4.0.3/4 versions) need to be explicitly updated. Such statistics suggest that potentially more than 4 out of 5 Android devices suffer from the Zygote vulnerability.

Since it is realistically impossible to assume that so many devices will be patched or upgraded by device vendors, proper controls should be performed at repositories of applications in order to recognize and block the spread of malicious applications like DoSChecker. The main distribution channels for Android applications are appstores and unknown source (e.g. forums). In general, Android appstores natively rely on cooperation among users to isolate bad/unsatisfying applications by means of feedback mechanisms. However, feedbacks are not a sufficient solution for security issues since the average user has not enough expertise to evaluate/recognize malicious behavior during the normal use of a mobile device\textsuperscript{17}.

For this reason, Google has recently proposed a security service for scanning applications, called Bouncer, in order to mitigate the diffusion of malicious application on Google Play. Despite this, Oberheide and Miller (2012) have discovered that such security mechanism can be easily bypassed, thus showing that the spreading of malicious applications through official repositories is far from being solved.

7.3. Security Analysis of other Android Sockets

Besides the Zygote socket, we discovered that two other sockets have access permissions set to 666 at the Linux layer: the keystore socket and the property_service socket.

\textsuperscript{16}http://developer.android.com/about/dashboards/index.html
\textsuperscript{17}As underlined also by test results performed in Sec.6
The **keystore** socket is used to communicate with the corresponding Keystore Service, which handles cryptographic keys and certificates for both the user and the system. The security of this socket is ensured by the owner itself. In fact, differently from the Zygote process, the Keystore Service uses the socket **credential passing mechanism** to obtain the identity (i.e. UID and GID) of the requester. Moreover, it contains a static list of users allowed to perform sensitive operations (like creating or deleting a keystore). Thus, commands sent by a malicious application (with an UID not belonging to the static list) are rejected. For the sake of completeness, we have also performed an automatic check with **DoSChecker**. Our malicious application, as expected, is able to connect, read and write on the socket but it is not able to perform any malicious operation.

The **property_service** socket is owned by the service with the same name, which runs in the **init** daemon. The Property Service is used to store system-wide properties (e.g. status of the GPS module) and to start/stop system services. Also in this case, the security is ensured by the owner of the socket. In particular the Property Service retrieves the identity of the requester from the socket (using the credential passing mechanism) and the property requested. This information is checked against a list that binds the allowed users with the properties they can handle. If no match is found a security exception is thrown and the operation is aborted. The **DoSChecker** analysis confirms that, although it is possible to connect to the socket, no malicious operations, like changing properties unexpectedly, can be carried out.

8. Related Work

The security of the Android platform is a quite a new research field. The literature on the topic can be classified into three trends: i) static analysis of Android applications, ii) assessment of Android access control and permissions policies, and iii) malware and virus detection.

Static analysis is concerned with the development of white box or black box techniques for analyzing the behavior of Android applications in order to assess whether their installation may jeopardize the security of the mobile device. To this aim, in Enck et al. (2011) a study of Android applications aimed at identifying those applications that steal personal data is presented. In Fuchs et al. (2010) Scandroid is proposed as a tool for automated security certification of Android applications. Scandroid checks whether data flows
among applications are consistent with security specifications extracted from their manifests.

Furthermore, a language-based approach to infer security properties from Android applications is proposed in Chaudhuri (2009). The same work proposes a type system that guarantees that well-typed programs respect user data access permission. In Armando et al. (2012a) a framework for supporting the formal modeling of the interactions occurring at different layers of the Android architecture is discussed.

Several works are focused on analyzing/extending the native Android access control model. In Shin et al. (2010) a formalization of the Android permission system based on a state-based model representing entities, relations and constraints is proposed. Furthermore, Felt et al. (2011) proposes a methodology for assessing the real privileges of Android applications. The same paper also proposes Stowaway, a tool for detecting over-privilege in compiled Android applications. Nauman et al. (2010) proposes Apex, a policy enforcement framework for Android that allows users to selectively grant permissions to applications as well as to impose constraints on the usage of resources. A different approach is followed in Ongtang et al. (2009). Such work proposes Secure Application INTeraction (SAINT), a modified infrastructure that governs install-time permission assignment. Other works are focused on issues related to privilege escalation. For instance, XManDroid (eXtended Monitoring on Android - Bugiel et al. (2011)) is a security framework that extends the native monitoring mechanism of Android to detect privilege escalation attacks. In Davi et al. (2011) the authors show that, under proper assumptions, a privilege escalation attack is possible in the Android framework. An interesting approach to the analysis of the native Android security policies is presented in Chin et al. (2011) where possible threats and solutions to mitigate the problem of privilege escalation are discussed. Since the vulnerability presented in our paper requires no dangerous permission, none of the approaches proposed in the above papers can detect or prevent it.

In Dagon et al. (2004) a comprehensive assessment of the state of the art of mobile viruses and malware for Android is performed, while in Zhou and Jiang (2012) the detection rate of four representative mobile security tools with a dataset of more than 1,200 malware samples is assessed.

Crowdroid, a malware detector executing a dynamic analysis of the application behavior, is proposed in Burguera et al. (2011) while in Schmidt et al. (2009) an automatic inspection of Android executables is performed in
In order to extract function calls and compare them with malware executables for classification. A similar approach is followed by the DroidRanger tool in Zhou et al. (2012), which combines a footprint-based detection engine of known malware with an heuristic detection engine for day-zero malware. In Mylonas et al. (2011) the feasibility of developing malwares for smartphone platforms by average programmers that have access to the official tools and programming libraries is discussed.

Current malware detection tools are unable to recognize the Zygote vulnerability. Of course, as soon as techniques for automatically detecting (e.g. through testing) this type of vulnerabilities will become available, repositories employing app testing can help in identifying and preventing the distribution of the malicious apps based on them.

Finally, some works have been driven by the need to improve the privacy of the users. In this direction, in Zhou et al. (2011) the need of a new native privacy mode in Android smartphones is discussed, while Fragkaki et al. (2012) proposes Sorbet, an enforcement system that allows developers to protect their applications against undesired information flows.

9. Conclusions

In this paper we analyzed the Android cross-layer architecture and the interactions between layers. We discussed the native Android security mechanisms and we showed that the flow responsible of launching applications in the Android OS contains a severe vulnerability that can be used by malicious applications to mount a fork bombing attack that makes the device totally unresponsive. In order to demonstrate the impact of this vulnerability we developed DoSChecker, a malicious application that by exploiting the vulnerability shows that all Android security mechanisms provide no effective protection. Besides reporting our findings, we proposed to the Android Security Team two fixes for patching the vulnerability. A slight adaptation of a proposed fix has been included in the main Android development from version 4.1, while one of our original patch have been adopted in a well-known customized Android distribution (namely the CyanogenMod) from version 4.0.3 on. Our analysis also indicates that other two sockets used by Android share many similarities with the Zygote socket but they do not suffer from the vulnerability we found in the Zygote socket.

Our work sheds light on an otherwise unexplored area of the Android security model, namely the interplay between the layers that lay at the core of
Android. As a continuation of the current work we have developed a detailed and formal account of the security of the Android cross-layer architecture (Armando et al. (2012a)). This work provides a firm basis for the formal and systematic analysis of the security-critical flows used in Android (e.g. all flows that require the switching from user to kernel mode) both for private and professional use (Armando et al. (2013)) of Android devices.
References


Oberheide J, Miller C. Dissecting the android bouncer. SummerCon 2012;URL: http://jon.oberheide.org/research/.


