Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some Countermeasures)*

Alessandro Armando[1,2], Alessio Merlo[1,3,**], Mauro Migliardi[4], and Luca Verderame[1]

[1] DIST, Università degli Studi di Genova, Italy
{armando,alessio.merlo}@dist.unige.it
luca.verderame@ai-lab.it
[2] Security & Trust Unit, FBK-irst, Trento, Italy
armando@fbk.eu
[3] Università e-Campus, Italy
alessio.merlo@uniecampus.it
[4] DEI, University of Padova, Italy
mauro.migliardi@unipd.it

Abstract. We present a previously undisclosed vulnerability of Android OS which can be exploited by mounting a Denial-of-Service attack that makes devices become totally unresponsive. We discuss the characteristics of the vulnerability – which affects all versions of Android – and propose two different fixes, each involving little patching implementing a few architectural countermeasures. We also provide experimental evidence of the effectiveness of the exploit as well as of the proposed countermeasures.

1 Introduction

With more than 45% of US sales of smartphones in 3Q2011 [1], Android is arguably one of the greatest success stories of the software industry of the last few years. By leveraging a generic (albeit optimized for limited resource consumption) Linux kernel, Android runs on a wide variety of devices, from low-end to top-notch hardware, and supports the execution of a large number of applications available for download both inside and outside the Android Market.

Since most of the applications are developed by third-parties, a key feature of Android is the ability to sandbox applications, with the ultimate objective to achieve 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.*
> `http://developer.android.com/guide/topics/security/security.html`

* This work has been partially founded by EU project FP7-257876 SPaCIoS.
** Corresponding author.

D. Gritzalis, S. Furnell, and M. Theoharidou (Eds.): SEC 2012, IFIP AICT 376, pp. 13–24, 2012.

Sandboxing is achieved combining the isolation guaranteed by the use of Virtual Machines together with the enforcement mechanism that can be obtained from the Linux access control by giving each application a different Linux identity (i.e. Linux user). Each of these two mechanisms is well known and has been thoroughly tested to achieve a high level of security; however, the interaction between them has not yet been fully explored and may still hide unchecked vulnerabilities.

In this paper we present a previously unknown vulnerability in Android OS that allows a malicious application to force the system to fork an unbounded number of processes and thereby mounting a Denial-of-Service (DoS) attack that makes the device totally unresponsive. Rebooting the device does not necessarily help as a (very) malicious application can make herself launched at boot-time. Thus, our findings show that the aforementioned Android design goal is not met: a malicious application can indeed severely impact all other applications, the operating system, and ultimately the user. To overcome this empasse we propose two solutions, each involving very small changes in the system, that fix the problem. We present experimental results confirming that all versions of Android OS (including the most recent ones, namely versions 4.0 and 4.0.3) suffer from the vulnerability. The experiments also confirm that our proposed fixes counter the DoS attack.

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.

Structure of the Paper. In the next section we provide a brief description of Android OS. In Section 3 we present the vulnerability and in Section 4 we illustrate two possible solutions. In Section 5 we present some experimental results that confirm the effectiveness of the DoS attack as well as of the proposed countermeasures. In Section 6 we investigate works related to the security of the Android platform. We conclude in Section 7 with some final remarks.

2 The Android Architecture

The Android Architecture consists of 5 layers. The bottom layer (henceforth the *Linux layer*) is the Linux kernel. On top of the Linux layer live four Android-specific layers that we collectively call the *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 on the top of the stack and comprise both user and system applications which have been installed and execute on the device.

- *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 component containing the main modules for managing the device (e.g. *Activity Manager* and *Package Manager*) and for interacting with the underlying Linux drivers (e.g. *Telephony Manager* and *Location Manager* that handle the mobile hardware and the GPS module, respectively)
- *Android Runtime Layer.* This layer comprises the Dalvik virtual machine, the Android's runtime's core component, specifically optimized for efficient concurrent execution of disjoint VMs in a resource constrained environment. The Dalvik VM executes application files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint.
- *Libraries Layer.* The Libraries layer contains a set of C/C++ libraries providing useful tools to the upper layers. They are widely used by Application framework services. Examples of libreries are *bionic libc* (a customized implementation of libc for Android) and *SQL lite*.

2.2 The Linux Layer

Android relies on Linux kernel version 2.6 for core system services. Such services include process management and Inter-Process Communication (IPC). Each Android application, together with the corresponding Dalvik VM, is bound to a separate Linux process. Android uses a specific process, called Zygote, to enable code sharing across VM instances and to provide fast start-up for new processes. The Zygote process is created during Linux boot-strap. Every time a new Linux process is required (e.g., for starting a new Android application), a command is sent through a special *Unix domain socket* called Zygote socket. The Zygote process listens for incoming commands on the Zygote socket and generates a new process by forking itself as a Linux process. Differently from what happens in a normal Unix system, specialization of the child process behavior is not obtained by loading a new executable image, but only by loading the Java classes of the specific application inside the VM.

Communication between apps in Android is carried out through Unix sockets or the Binder driver. Sockets are used when data are small and well codified such as in Zygote socket case. In other cases (e.g., when data are big and heterogeneous), the Binder driver is used. Binder IPC mechanism relies on a kernel driver. Each process registers itself to the Binder driver and gets back a file descriptor. A process that wants to communicate with another process can simply send data through the file descriptor via an IOCTL command. The Binder kernel module sends received data directly to the destination process.

2.3 Interaction between the Android Layers and the Linux Layer

The interaction between the Android and the Linux layers is depicted in Fig. 1. When an application is launched a `startActivity` call is sent to the *Activity Manager Service*, a part of the System Server. The *Activity Manager Service*

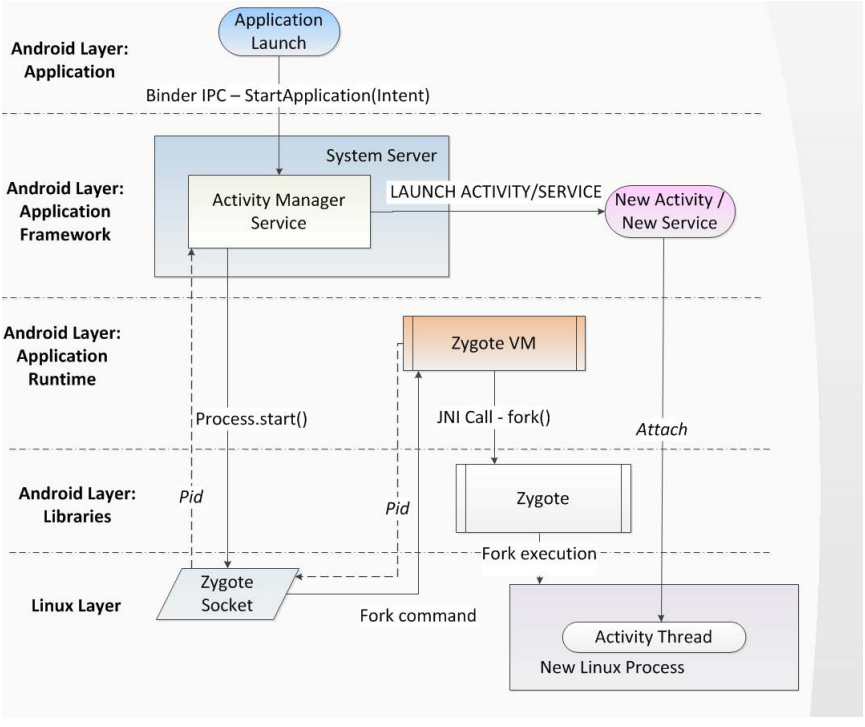

Fig. 1. Creation of a new process in Android

determines if the 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; thus, the Activity Manager Service gets the corresponding process and brings back the application to foreground.

In the latter case, 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 Linux layer via the Zygote socket and sends the fork command. The Zygote process has the exclusive right to fork new processes, thus, when Android requires the creation of a new process, the command must be issued to the Zygote process through the Zygote socket.

The command sent to the Zygote socket includes the name of the class whose static `main` method will be invoked to specialize the child process. The System 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 attempted. If no application is available for binding, the same class asks the Linux layer 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.

3 The Vulnerability

The Zygote socket is owned by `root` but has permissions 666 (i.e. `rw-rw-rw`), implying 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 process hosting the System Server (whose UID is 1000, it is not owned by root, nor it belongs to the root group) to request the spawning of new processes. However, this has the side effect to enable any process to ask for a fork.

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:

```
1  boolean runOnce() throws ZygoteInit.MethodAndArgsCaller
       {
2      ...
3        applyUidSecurityPolicy(parsedArgs, peer)
4        applyCapabilitiesSecurityPolicy(parsedArgs, peer);
5        applyDebuggerSecurityPolicy(parsedArgs);
6        applyRlimitSecurityPolicy(parsedArgs, peer);
7      ...
```

The policy prevents from

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

Moreover, only few limitations are put on the (static) class used to customize the child process. In particular, two checks are performed namely checking whether 1) the class contains a static `main()` method and 2) it belongs to the System package, which is the only one accepted by the Dalvik System Class loader.

Unfortunately, these security checks do not include a proper control of the identity (i.e., UID) of the requestor, therefore allowing each Linux process (and, its bound Android application or service) to send not necessarily legitimate but acceptable in the current Android security framework fork command to the Zygote socket 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, generating a dummy process which is kept alive at Linux layer. Such class does not perform any binding operation with an Android application, thus not triggering the removal of unbound new processes as the default `android.app.ActivityThread` class does.

In this way, all the security policies applied by the Zygote process are bypassed, leading to the building of a persistent Linux process which occupies memory resources in the device.

Thus, by flooding the Zygote socket with such requests, an increasingly large number of dummy processes is built until all the memory resources are exhausted (Fork bomb attack).

The Android layers are unable to notice the generation of dummy processes and, consequently, to intervene.

On the other hand, the creation of processes at the Linux layer is legal and managed directly by the kernel. Thus, none of the involved layers 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 bootstrapping, it is possible to lock the device into an endless boot-loop, thereby locking the use of the device.

Notice that to mount the attack, the malicious application does not require any special permission, therefore it looks harmless to the user upon installation.

4 Countermeasures

We describe 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 our patch is trivially adaptable to future developments).
2. **Zygote socket fix**. This fix restricts the permissions on the Zygote socket at the Linux layer.

4.1 Checking Fork Requests Inside the Zygote Process

As said in Section 3, the Zygote process does not perform any specific check on the identity of the process that requests the fork operation. Nevertheless, the Zygote socket is a Unix Domain socket created during the boot-strap of the Linux system. An important feature of Unix domain sockets is the *credential passing mechanism* which allows to identify endpoints connected to the socket by means of their PID, 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 process corresponding to the System Server 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`):

```
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.

4.2 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—as we said above—it has permissions 666. It is possible to modify both the owner (no `root`) and permissions of Zygote socket from 666 (i.e., `rw-rw-rw`) 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. Create a new owner for the Zygote socket. We added a new UID (i.e. 9988) in the file `android_filesystem_config.h` which contains statically assigned UID for Android system services. Then, in the same file we bind an ad hoc user `zygote_socket` and the new UID.

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

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

```
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. Include the UID of the Zygote socket owner in the group list of the System Server. Since also the System Server is 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:

```
...
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 modification implies that user applications cannot connect to the Zygote socket while the System Server (which is in the Zygote socket's group) can still issue the fork command.

5 Experimental Results

We tested the DoS vulnerability on all versions of Android OS currently available (i.e. $\leq$ 4.0.3) by building an Android malicious application (i.e. DoSChecker) as described in Sect. 3. We made tests both on actual and simulated devices. All our tests have produced an unbounded number of dummy processes, thus revealing that all versions 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 Tab. 1; newer Android versions, like 4.0

and 4.0.3, have been tested, instead, on Android device emulator. The behavior of the actual and simulated devices have been traced with *Adb Logcat* tool and *Adb shell* via a Windows shell.

Since a DoS is strictly related to the amount of resources, we tested actual devices with heterogeneous hardware in order to assess the relation between the availability of resources and the time spent to accomplish a successful attack.

Table 1. Actual devices used in experiments

Device Model	Android Versions
Lg Optimus One p550	2.2.3, 2.2.4 (stock LG), 2.2.4 (rooted)
Samsung Galaxy S	2.3.4 (stock Samsung), 2.3.7 (rooted Cyanogen 7)
Samsung Next GT-S5570	2.3.4 (stock Samsung), 2.3.4 (rooted)
Samsung Galaxy Tab 7.1	3.1 (stock Samsung), 3.1 (rooted)
HTC Desire HD	2.3.2 (stock HTC), 2.3.2 (rooted)

5.1 Exploiting the Vulnerability

Testing with Actual Devices. Once activating the DoSChecker application, 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 from the amount of resources available in the device. During the attack, users experience 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 has no access to the dummy processes. This behavior leads to the killing of other application's processes including system processes (such as home process). In several cases also the *System Server* crashes.

Once an application is killed, Android tries to restart it after a variable period of time but, in such scenario, 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 restarts applications the device is restarted. 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 provided us a twofold opportunity. First, the Android emulator allowed us to check the DoS vulnerability on newer Android versions such as Android 4.0 and 4.0.3; the testing procedure is similar and the experimental result is still 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 amount of dummy processes generated would greatly overcome the hardware capability of any device currently available on market.

Running DoSChecker as a Boot Service. Since the exploitation of the vulnerability takes to the reboot of the device, we added DoSChecker as a service starting at boot. Such operation is pretty straightforward and it does not require any specific Android permission. In particular, we added a java class in DoSChecker that, acting as a Broadcast receiver, intercepts the system broadcast messages and starts DoSCheck as a service whenever a bootstrap terminates (i.e. when the system broadcast message `android.intent.action.BOOT_COMPLETED` is received). Our tests show that the exploitation of the vulnerability at boot prevents Android from successfully completing the reboot process, thus getting the mobile device stuck. The only ways to recover the phone in such scenario, are to identify and manually uninstall the malicious application via *adb tool* or reflashing the device.

5.2 Testing the Countermeasures

We implemented the two countermeasures (i.e. Zygote process and Zygote socket fixes) proposed in Sect. 4. In particular, for each Android version, both on the actual and emulated devices, we built two patched versions, each implementing a countermeasure, by recompiling Android from scratch. The building process provides two output images called `system.img` and `ramdisk.img`. `System.img` contains all the Android system classes compiled while `ramdisk.img` corresponds to the Android RAM disk image which is loaded during bootstrapping and which 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 needs a modification of `init.rc`, affects both `system.img` and `ramdisk.img`.

Our tests show that both countermeasure are effective and prevent the fork of dummy process, thereby solving the vulnerability. Moreover, by using actual devices we have empirically proven that the proper functioning of devices was preserved (i.e. the proposed countermeasures do not affect the normal flow of Android) in all the tested cases.

6 Related Works

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

Static analysis is related to the development of white box or black box methodologies for detecting malicious behaviors in Android applications before installing them on the device. To this aim, Enck et al. [2] executed a horizontal study of Android applications aimed at discovering stealing of personal data. Besides, Fuchs et al. [3] proposed Scandroid as a tool for automatically reasoning about the security of Android applications.

Static analysis could help identifying the calls to the Zygote socket. However, UNIX sockets might be used for legitimated goals and recognizing the specific socket name might be made problematic even with a simple string concatenation.

In any case, to our knowledge no current static analysis tool identifies exploits of the described vulnerability.

The main part of current literature is focused on access control and permissions. For instance, [4] proposes a methodology for assessing the actual privileges of Android applications. This paper also proposes Stowaway, a tool for detecting over-privilege in compiled Android applications. Nauman et al.[5] proposes Apex as a policy enforcement framework for Android that allows a user to selectively grant permissions to applications as well as to impose constraints on the usage of resources. A different approach is proposed by Ongtang et all [6] who present Secure Application INTeraction (SAINT), a modified infrastructure that governs install-time permissions assignment. Other works are focused on issues related to privilege escalation. Bugiel et al. [7] propose XManDroid (eXtended Monitoring on Android), a security framework that extends the native monitoring mechanism of Android to detect privilege escalation attacks. In [8] authors state that, under proper assumptions, a privilege escalation attack is possible on Android framework. Some research were made on analyzing native Android security policies [9] focusing on possible threats and solutions to mitigate the problem [10] of privilege escalation.

Nonetheless, the vulnerability disclosed in this paper requires no special permission, thus these approaches are not a valid solution.

Regarding virus and malware detection, Dagon et al. [11] made a comprehensive assessment of the state of the art of mobile viruses and malwares which can affect Android devices. In [12] a methodology for mobile forensics analysis in Android environments is proposed. In [13] Crowdroid, a malware detector executing a dynamic analysis of the application behavior, is proposed. Schmidt et al. [14] inspect Android executables to extract their function calls and compare them with malware executables for classification aim.
Specific malware signatures for exploiting the vulnerability described in this paper could be generated, however none is available today.

Besides, from a general point of view, all these works have been driven by the need to improve the privacy of the user. In such direction, Zhou et al. [15] argue the need of a new native privacy mode in Android smartphones.

At the best of our knowledge, this is the first work related to Denial of Service issues on Android platform.

As a final remark, none of the previous works investigates relations and security issues related to interactions between the Android and the Linux layers.

7 Conclusions

In this paper, we presented a previously undisclosed vulnerability on Android devices which is the first vulnerability on Android that leads to a DoS attack of this severity. We also developed a sample malicious application, (i.e. DoSCheck) which exploits the vulnerabilities, and we proposed two fixes for securing the Android OS against the vulnerability. We reported such vulnerability to Android security team which will include a patch in an upcoming update of the Android OS. Furthermore, we plan to publicly release both DoSCheck code and patched

systems in the very near future, accordingly with a responsible disclosure policy we are discussing with Android group and Open Handset Alliance.

References

1. Gartner Group. Press Release (November 2011), http://www.gartner.com/it/page.jsp?id=1848514
2. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, p. 21. USENIX Association, Berkeley (2011)
3. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications
4. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 627–638 (2011)
5. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, pp. 328–332. ACM, New York (2010)
6. Ongtang, M., Mclaughlin, S., Enck, W., Mcdaniel, P.: Semantically rich application-centric security in android. In: ACSAC 2009: Annual Computer Security Applications Conference (2009)
7. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt (April 2011)
8. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege Escalation Attacks on Android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
9. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys 2011, pp. 239–252. ACM, New York (2011)
10. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S.: Google android: A state-of-the-art review of security mechanisms. CoRR, abs/0912.5101 (2009)
11. Dagon, D., Martin, T., Starner, T.: Mobile phones as computing devices: The viruses are coming! IEEE Pervasive Computing 3(4), 11–15 (2004)
12. Di Cerbo, F., Girardello, A., Michahelles, F., Voronkova, S.: Detection of Malicious Applications on Android OS. In: Sako, H., Franke, K., Saitoh, S. (eds.) IWCF 2010. LNCS, vol. 6540, pp. 138–149. Springer, Heidelberg (2011)
13. Burguera, I., Zurutuza, U., Nadjm-Therani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011 (2011)
14. Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yuksel, K.A., Camtepe, S.A., Albayrak, S.: Static analysis of executables for collaborative malware detection on android. In: IEEE International Conference on Communications, ICC 2009, pp. 1–5 (June 2009)
15. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming Information-Stealing Smartphone Applications (on Android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) Trust 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)