Google Android: A Comprehensive Security Assessment

Article *in* IEEE Security and Privacy Magazine · May 2010

DOI: 10.1109/MSP.2010.2 · Source: IEEE Xplore

Shlomi Dolev
Ben-Gurion University of the Negev
292 PUBLICATIONS **4,546** CITATIONS
SEE PROFILE

Chanan Glezer
Ariel University
34 PUBLICATIONS **1,232** CITATIONS
SEE PROFILE

Google Android: A Comprehensive Security Assessment

This research provides a security assessment of the Android framework—Google's software stack for mobile devices. The authors identify high-risk threats to the framework and suggest several security solutions for mitigating them.

Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer
Ben-Gurion University of the Negev, Israel

As an operating system for mobile devices, Google's Android—an open, programmable software framework—is vulnerable to typical smart-phone attacks. Such attacks can make the phone partially or fully unusable, cause unwanted SMS/MMS (short message service/multimedia messaging service) billing, or expose private information.[1] Smart-phone attack vectors include cellular networks, Bluetooth, the Internet (via Wi-Fi, General Packet Radio Service/Enhanced Data Rates for Global Evolution, or 3G network access), USB, and other peripherals.[2] Smart-phone malware evolve very quickly due to the experience malware writers have gained with PC malware. As Alexander Gostev estimates,[3] two years are sufficient for smart-phone viruses to evolve to a level that computer viruses only reached after 20 years. Thus, the challenge in ensuring smart-phone security is becoming similar to that confronting the PC.[4] Among the most prominent smart-phone malware types are the Lasco/Cabir[5] and Commwarrior/Mabir worm families,[6] the FlexiSpy, RedBrowser, and Skulls Trojan horses, the WinCE.Duts and CardTrap viruses,[7] and, recently, the iPhone ikee worm and iPhone/Privacy.A hacking tool that exploited vulnerabilities in jail-broken iPhone systems.

So far, smart-phone pandemic outbreaks have been limited in their scale and impact due to a restricted number of potential victims. Nevertheless, smart-phone market share in the US has increased from 11 percent of all cellular phone subscribers in 2008 to 17 percent in 2009, and it is expected to increase significantly over the next few years.[8] Consequently, smart phones are likely to become a fertile ground for various types of malware.

The number of smart phones based on the Google Android operating system are expected to increase 10 percent during 2010.[8] The risks to Android are nevertheless significant, mainly because it's an open source software stack operated in a heterogenic mobile environment. Thus, hackers can more easily access, manipulate, and exploit the operating system code. Here, we review and assess the security mechanisms incorporated into Google's new Android framework and whether they're adequate in light of the emerging threats to smart phones.

Android Security Mechanisms

Android is an application execution environment for mobile devices. It includes an operating system, application framework, and core applications. The Android software stack is built on the *Linux kernel*, which is used for its device drivers, memory management, process management, and networking. The next level up contains the *Android native libraries*. Various system components in the upper layers use these libraries, which are written in C/C++. Incorporating these libraries in Android applications is achieved via Java native interfaces. The next level is the Android *runtime*, comprising the *Dalvik virtual machine* and the *core libraries*. Dalvik runs `.dex` (Dalvik-executable) files that are designed to be more compact and memory-efficient than Java class files. The core libraries are written in Java and provide a substantial subset of the Java 5 SE packages as well as some Android-specific

Table 1. Security mechanisms incorporated in Android.

Mechanism	Description	Security issue
Linux mechanisms		
POSIX users	Each application is associated with a different user ID (or UID).	Prevents one application from disturbing another
File access	The application's directory is only available to the application.	Prevents one application from accessing another's files
Environmental features		
Memory management unit (MMU)	Each process is running in its own address space.	Prevents privilege escalation, information disclosure, and denial of service
Type safety	Type safety enforces variable content to adhere to a specific format, both in compiling time and runtime.	Prevents buffer overflows and stack smashing
Mobile carrier security features	Smart phones use SIM cards to authenticate and authorize user identity.	Prevents phone call theft
Android-specific mechanisms		
Application permissions	Each application declares which permission it requires at install time.	Limits application abilities to perform malicious behavior
Component encapsulation	Each component in an application (such as an activity or service) has a visibility level that regulates access to it from other applications (for example, binding to a service).	Prevents one application from disturbing another, or accessing private components or APIs
Signing applications	The developer signs application `.apk` files, and the package manager verifies them.	Matches and verifies that two applications are from the same source
Dalvik virtual machine	Each application runs in its own virtual machine.	Prevents buffer overflows, remote code execution, and stack smashing

libraries. The *application framework* layer, written fully in Java, includes Google-provided tools as well as proprietary extensions or services. The topmost *application* layer provides applications such as a phone, Web browser, email client, and more.

Each application in Android is packaged in an `.apk` archive for installation. This archive is similar to a Java standard `jar` file in the way that it holds all code and noncode resources (such as images or manifest) for the application. Android applications are written in Java based on the APIs the Android software development kit (SDK) provides. William Enck and his colleagues discussed the main components of an Android application and how to use an Android-specific mechanism to protect Android applications.[9]

In general, several security mechanisms are incorporated into the Android framework (see Table 1). We can cluster them into three general groups: Linux mechanisms, environmental features, and Android-specific mechanisms.

Linux Mechanisms

There are two primary security mechanisms at the Linux layer in Android: the Portable Operating System Interface (POSIX) users, and file access. The basic element of these mechanisms is *users* (that is, entities). Each *object* (such as a process or file) is owned by a user (represented by an integer number, or *user id*). Users are further assigned to groups.

POSIX users. Each Android package (`.apk`) file installed on a device has a unique Linux (POSIX) user ID. Thus, two different packages' code can't run in the same process. In a way, this creates a sandbox and prevents applications from interfering with one another. For two applications to share the same permissions set and possibly run in the same process, they must share the same user ID, which is possible only using the sharedUserID feature. To share the same user ID, two applications must explicitly declare they're using the same sharedUserID, and both must be digitally signed by the same author.

File access. Files in Android (both application and system files) are subject to the Linux permissions mechanism. Each file is associated with the owner's user ID and group ID and three tuples of `Read`, `Write`, and `eXecute` (`rwx`) permissions. The Linux kernel enforces these permissions and imposes the first tuple on the owner, whereas the second affects users that belong to the owner's group, and the third affects the rest of the users.

Generally, either the "system" or "root" user owns system files in Android, whereas an application-specific user owns application files. Assigning different users for each application and for the system files along with proper permissions settings provides the needed security for file access. These actions are necessary because application files won't be accessible to other applications (unless they're using the sharedUserID feature or have been set as globally readable/writeable).

Additionally, Linux handles many system functionalities as pseudo-files. Consequently, the file access mechanism effectively allows setting permissions on drivers, terminals, hardware sensors, power state changes, audio, direct input readings, and so on.

Environmental Features

The surrounding technological environment (for example, the hardware, programming language, and mobile carrier's infrastructure) provides several mechanisms that enhance the Android device's security.

Memory management unit. A prerequisite for many modern operating systems, Linux in particular, is that they have a *memory management unit* (MMU), a hardware component that facilitates separating processes into different address spaces (virtual memory). Various operating systems employ the MMU to ensure that one process can't read another's memory pages (information disclosure) or corrupt them. This reduces the likelihood of privilege escalation because a process can't make its own code run in a privileged mode by overwriting the private operating system memory.

Type safety. Type safety is a programming language property that forces variable contents to adhere to a specific format and thus prevents erroneous or undesirable usage. Partial or missing type-safety or boundary checks can lead to memory corruption and buffer-overflow attacks, which, in turn, provide the means for arbitrary code execution.

As noted, Android employs Java, a strongly typed programming language. Programs written in Java are less susceptible to arbitrary code execution, whereas languages such as C allow casting without type checking and perform no boundary checks unless the programmer specifically includes them. Android does allow programs with native components written in C, at the risk of potentially reduced security.

Binder, the Android-specific *inter-process communication* (IPC) mechanism, is also type-safe. Developers define the types of data elements that Binder passes, using the Android Interface Definition Language (AIDL), at compile time. This ensures that types are preserved across process boundaries.

Mobile carrier security features. Telephony systems have a basic set of attributes and functionalities stemming from a need to identify users, monitor usage, and charge the client accordingly. A more general term for these features is AAA (authentication, authorization, and accounting). As a smart-phone platform, Android borrows these classical security features from cellular phone design. Authentication usually occurs via SIM card and associated protocols.

Android-Specific Security Mechanisms

Android provides the following dedicated security mechanisms introduced by Google: application permissions, component encapsulation, and signing.

Application permissions. Android's permissions mechanism for applications enforces restrictions on specific operations that an application can perform. Android has roughly 100 built-in permissions that control operations ranging from dialing the phone (`CALL_PHONE`), taking pictures (`CAMERA`), using the Internet (`INTERNET`), listening to key strokes (`READ_INPUT_STATE`), and even disabling the phone permanently (`BRICK`). Any Android application can declare additional permissions. To obtain a permission, an application must explicitly request it. Permissions have associated protection levels:

- *normal* (application-level permissions not especially dangerous to possess);
- *dangerous* (a high-risk permission that might provide access to private data or harmful functionalities; an application can't receive such permissions without the user's explicit confirmation);
- *signature* (a permission that can be granted only to other packages that are signed with the same signature as the one declaring the permission); and
- *signature-or-system* (a special type of signature permission that's also granted to any package installed in the system image).

At installation, the system grants permissions that the installed application requests based on checks of that application's signature against those of the applications declaring those permissions (for signature or signature-or-system protection-level permissions) and on the user's approval (for normal or dangerous protection-level permissions). After the user has installed the application and it receives its permissions, it can no longer request any more permissions. An operation that hasn't received permission at installation time will fail at runtime.

Component encapsulation. An Android application can encapsulate its components within the application content. This prevents other applications from accessing them, assuming that they bear a different user ID. This occurs primarily through the definition of the component's "exported" property. If the "exported" property is set to "false," the component at hand can be accessed only by the application that owns it (and any other application that shares the same user ID through the sharedUserID feature). If set to "true," external entities can access it. These entities are controlled through permissions as described in the previous section. Developers should always set the "exported" property manually because the default encapsulation policy might not coincide with the expected one.

Signing applications. Each application in Android is packaged in an `apk` archive for installation. The Android system requires that all installed applications be digitally signed (code and noncode resources). The signed `apk` is valid as long as its certificate is valid and the enclosed public key successfully verifies the signature. Signing applications in Android verifies that two or more applications are from the same author ("same-origin" verification). The sharedUserId mechanism and the permission mechanism use this method to verify signature and signature-or-system protection-level permissions.

Android Framework Security Assessment

We performed a comprehensive assessment of various security aspects in the Android framework using HTC's G1 smart phone. Our assessment methodology included a code review of various Android components, analysis of applications' permission-granting mechanisms and the application-installation process, and a feasibility assessment of existing Linux and Java malware.

An Android device in its normal state is well-guarded given that an attacker (or the legitimate owner, for that matter) can replace neither the core components nor the kernel without manipulating the hardware, which is difficult. The only way to alter operating system components is to identify a vulnerability in one of the kernel modules or core libraries that will let the attacker acquire root access.

Whenever an attacker discovers a bug or vulnerability in a core component, he or she might be able to run malicious code in a highly privileged mode and even gain full control over the device. This threat is amplified because some system processes run with root privileges, no fine-grained access control mechanism exists for system processes, and Android's source code is publicly available. On the other hand, this public availability provides certain benefits—among them, that multiple individuals can check and verify the code. Even if the platform is never fully secure, the open source approach promotes steady, ongoing security improvements. As time passes, we expect the number of bugs to diminish and the system to become less vulnerable. All in all, we believe that attackers exploiting a bug or vulnerability in a core component is a likely scenario.

To remotely attack an Android device, a vulnerable service must be exposed on the Internet. This requirement reduces the likelihood of remote attack scenarios because, by default, none of the services it's running are listening for incoming connections. The amount of exploitable code running on a device through local services, device drivers, and kernel code makes host-based exploitation attempts a higher risk. Thus, we regard the device as more vulnerable to local host-based exploitation attempts.

Another problem is that the permissions mechanism isn't sufficiently protected, and installing an application that maliciously uses permissions that an unaware user grants is a likely scenario. The framework also provides the `adb` install feature, which allows for installing applications and granting the permissions without any user interaction. In addition, a user can't approve a subset of requested permissions (it's "all or none") or verify that an application uses its granted permissions only for benign purposes. Moreover, the sharedUserID mechanism lets applications share permissions without a user's awareness or the need for explicit approval.

We also found that an attacker can maliciously inject code via a Web browser. WebKit, Android's open source Web engine, has a history of such vulnerabilities. Some recent attacks on it include a buffer overflow in an outdated native library and an explicit cross-site scripting (XSS) vulnerability. Both attacks let the attacker run malicious code on the device, with abilities and privileges assigned to the browser application.

Injecting malicious applications via Bluetooth is, however, not likely to occur because several protection mechanisms exist:

- a device can set the Bluetooth connection as not discoverable;
- if the Bluetooth connection is set as discoverable, it's only for two minutes;
- the owner needs to accept the connection; and
- the owner needs to manually install the file.

Our security inspection indicated that the system is well-protected against Structured Query Language (SQL) injection attacks. However, some information is fully exposed to attackers (for example, the content on the SD card). Having separate user IDs for each application protects the system and applications from tampering.

Figure 1 presents the results of a qualitative risk analysis we conducted to identify and prioritize the threats to which an Android device might be exposed. We based our evaluation on assessing the impact and likelihood of various threats (see the sidebar on p. 43

for a detailed taxonomy) that exploit vulnerabilities in the Android framework to harm, disable, or abuse the confidentiality, availability, or integrity of the following Android assets:

- private/confidential content that's stored on the device (pictures, contacts, email, documents, and so on);
- applications and services (phone and SMS applications, Internet, and email);
- resources such as battery power, communication, memory, and processing power (CPU); and
- hardware, including the device itself, external memory cards, the battery, and the camera.

We mapped the most important threats in relation to three levels of likelihood—unlikely, possible, and likely—and three optional impact values—minor, moderate, and severe.

Following these conclusions and the risk assessment, we identified the five most important threat clusters that Android should counter by employing proper security solutions and capabilities. We obtained these threat clusters by clustering similar threats assigned with the highest risk. We describe them in detail in the next section.

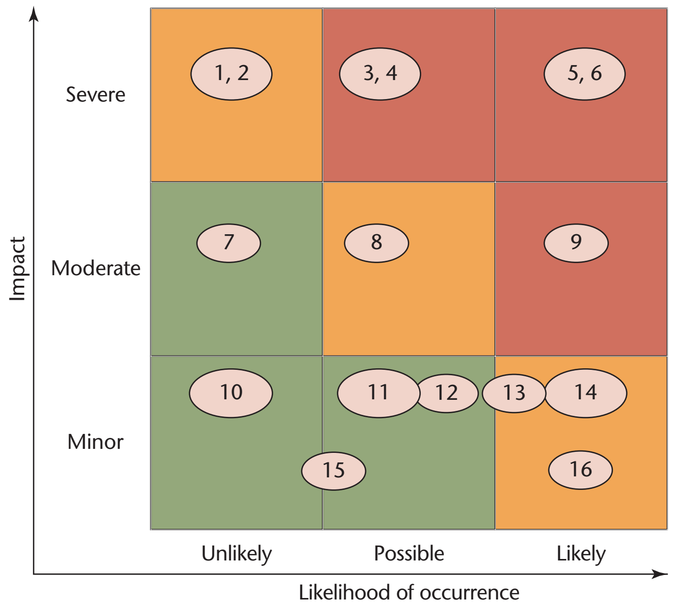

Figure 1. Qualitative risk analysis of threats (see the sidebar on p. 43 for a detailed taxonomy). Each threat is mapped to one of three levels of likelihood—unlikely, possible, and likely—and to one of three optional impact values—minor, moderate, and severe.

Security Solutions for Mitigating Android's Threats

Several security companies have already announced their intention of porting their security solutions to Android. For example, in 2008, SMobile released a security solution for Android-based handsets that includes antivirus and antitheft applications. Savant Protection, which specializes in intrusion prevention, announced in March 2008 that it ported its security solution, "Savant Technology," to Android. Mocana, another company that ported its solution to Android, claims that it offers the following capabilities: a secure browser; virtual private network (VPN) clients; malware protection; secure firmware update and secure boot capabilities; and certificate-handling features to authenticate devices, network services, and individuals to each other. DroidHunter is another security application available for Android. We can assume that companies that already have security solutions for mobile devices (such as Symantec, F-Secure, McAfee, and CheckPoint) will also port those to Android-based devices. Among the features these solutions include are antivirus software and intrusion-detection systems (IDSs) that run on the devices themselves.

To further harden an Android device and reduce the harmful potential of identified high-risk threats, users can employ several additional safeguards. We tested and evaluated some of these safety measures in our mobile security laboratory. For example, we ported a SELinux into Android and activated a security policy for enhancing the protection of system processes.[10] Moreover, we enabled a NetFilter-based firewall that users can easily configure via an Android-compliant interface. We also started investigating methods for locking an Android handset, such as with a password or other similar mechanism. In addition, we're developing and evaluating an IDS based on anomaly detection (called Andromaly). Table 2 summarizes applicable security solutions and the current state of market offerings.

Using the aforementioned five threat clusters, we assessed the mitigation level and effort required to apply various countermeasures for each cluster (see Table 3).

Threat Cluster 1

The first threat cluster compromises availability, confidentiality, or integrity by maliciously using the permissions granted to an installed application. This attack scenario is likely to happen and has a potentially high impact on the device. There are several possible countermeasures.

Intrusion-detection/prevention system. An IDS solution is well-suited for defining the normal behavior of the system, application, or user to detect deviations or, alternatively, malware behavioral patterns. An IDS can also serve as an effective tool in discovering initially unknown and isolated threats. However, because malware can quickly adapt and mask its behavior according to the security tools that can detect it, an IDS's effectiveness might decrease over time. Based

Table 2. Security mechanisms applicable to Android.

Mechanism	Description	Security issue	Existing tools
Antimalware	Scans files, memory, short message service (SMS), multimedia messaging service (MMS), email, URLs, and Java scripts	Viruses, Trojan horses, worms, root-kits, and other malware	SMobile, Mocana, DroidHunter, ClamAV*
Firewall	Can block or audit unallowed connections to or from the device	Services that are exposed to an untrusted network; network attacks	SMobile, Netfilter/iptables
Intrusion-detection/ prevention systems	Detects abnormal or known malicious behavior in the system, process, network traffic, or user	Fraud (for example, expensive calls), unusual telephone activity, theft, malicious attacks	Andromaly, DroidHunter
Linux access control	Limits the access of processes and users to resources or services	Damage from malicious or exploited applications	SELinux[10]
Login	Lets users provide a secret password to use the device	Unauthorized device use	Android screen-lock pattern
Selective Android permissions	Lets users grant only a subset of permissions to an installed application	Unneeded permissions that attackers can maliciously exploit	
Android permissions access control	Hardens Android devices by limiting granted permissions using a predefined policy; relevant mainly to corporate users	Unneeded permissions that attackers can maliciously exploit	Secure Application INTeraction†
Permissions management application	Scans Android's applications' permissions, giving the user a concise summary	Installed unwanted applications, Trojan horses	
Data encryption	Encrypts the device's content	Access to sensitive information when the device is lost or stolen	
Phone call encryption	Provides secured connection (authenticated or encrypted)	Eavesdropping, identity verification	
Spam filter	Blocks MMS, SMS, emails, and calls from unwanted origins	Spam	
Virtual private network	Connects to a remote network over the Internet; relevant mainly to corporate users	Insecure network connections	PPTP, L2TP, and IPSec-based VPN connections enabled on Android release 1.6 (that is, Donut)
Application certification	Allows for signing each application with a certificate authority (CA)	Damage from untrusted applications	Open Mobile Terminal Platform's (OMTP) Application Security Framework‡
Resource management	Enables fairness in resource allocation (CPU for phone application, disk quotas, I/O rate limiting and quotas, network quotas, and traffic shaping)	Denial of service (DoS)	
Remote management	Remotely configures and manages the device (settings, firewall policy, remote "bricking," application tracking); relevant mainly to corporate users	Device theft	
Context-aware access control	Dynamically allows and restricts access to resources and services based on a predefined model	Breaches of confidential content and the integrity of services	"Local" application on the Android Market, Andromaly
Integrity checking	Verifies system and application state	Offline tampering	

* A.D. Schmidt et al., "Enhancing Security of Linux-Based Android Devices," *Proc. 15th Int'l Linux Kongress*, Lehman, 2008.

† M. Ongtang et al., "Semantically Rich Application-Centric Security in Android," *Proc. 25th Ann. Computer Security Applications Conf.* (ACSAC 09), IEEE Press, 2009, pp. 340–349.

‡ *Open Mobile Terminal Platform (OMTP)*, Application Security Framework, June 2008.

Table 3. Countermeasures for tackling high-risk threats to Android.

	Solution	Mitigation level	Effort
Threat cluster 1: maliciously using the permissions granted to an installed application	Host-based intrusion-detection system (HIDS)	◐	Medium
	Firewall	◐	Low
	Application certification	●	High
	Selective Android permissions	◐	Low
Threat cluster 2: exploiting a vulnerability in the Linux kernel or system libraries	SELinux	◕	Low
Threat cluster 3: exposing private content	Login	◕	Low
	Firewall	◕	Low
	Data encryption	●	Low
	Context-aware access control	◕	Medium
	Remote management	◕	Medium
Threat cluster 4: draining resources	Resource management	●	High
	HIDS	◐	Medium
Threat cluster 5: compromising internal/protected network	Virtual private network	●	Low
	Remote management	●	Medium
	Context-aware access control	◕	Medium

on our implementation of the Andromaly IDS we developed for Android, we classify the development effort as medium because it requires system modification for more advanced capabilities.

Firewall. A firewall is a solution for network-related attacks that can prevent data leakage via malware that's already been installed. However, given that not all attacks that abuse permissions are network-based, a firewall might also be useful against a partial set of attacks. Because the NetFilter module that provides firewall capabilities at the Linux-kernel level is enabled on the HTC G1, only a control application is needed. We've developed a preliminary control GUI for managing these capabilities, based on the intermediate "iptables" command-line utility. Consequently, we can classify the effort needed to develop such an application as low.

Application certification. Certification is an ideal countermeasure against malicious applications. Because each application would have to be thoroughly tested and reviewed prior to certification and permission to use any feature of the device, malicious applications would certainly be caught in their early phases and be unable to receive the proper certification.

Unfortunately, nothing comes without a cost; with application certification, the cost of verifying every application is quite high.

Selective Android permissions. Letting the user approve only a subset of permissions to an installed application would reduce the risk of an application maliciously using granted permissions. This solution requires modifying and adding an advanced feature to the package installer activity so that the user could decline a particular requested permission. This wouldn't interfere with installing the application. Such a change would yield a high gain for security-aware users and wouldn't degrade usability for unaware users. It would also protect against granting unnecessary permissions. Overall, the required effort is low but necessitates a system modification and possible design changes.

Threat Cluster 2

The second threat cluster occurs when an application exploits a vulnerability in the Linux kernel or system libraries, thus compromising availability, confidentiality, or integrity. This scenario was proved possible, and our security analysis showed that additional vulnerabilities are likely to be found. Although this cluster of threats has a low probability of occurring, such threats could inflict severe damage.

The primary countermeasure to this threat is the SELinux solution. SELinux is well-suited to limiting the abilities of operating system entities. The hazardous potential of exploitable vulnerabilities could lead to a situation in which the whole could be undermined if the attacker obtained super-user privileges. By limiting the abilities of root processes and otherwise potentially vulnerable or high-priority entities, SELinux would prevent the attacker from forcing the system to do his or her bidding and so render the attack much less effective. However, because each entity must be able to execute certain commands for normal operation, SELinux can't block these commands. If the entity has been compromised, the attacker would still have some maneuvering space for unleashing an attack. In other words, the attack can only be partially deflected. Our experimentation with SELinux on Android has shown that it consumes very few resources, incurs low overhead on the system, and requires a low effort to apply.[10] The only issue to consider is creating a proper SELinux policy.

Threat Cluster 3

The third cluster of threats compromises the availability, confidentiality, or integrity of private or confidential content. Any application can read the SD card's contents, and attackers can eavesdrop on wireless communication remotely. There are several solutions to this threat.

Login. The requirement to enter a password or draw pattern to unlock certain device functionalities is a well-known and effective tool against various threats—in particular, the exposure of private content. If an attacker steals a device with the lock in place, he or she couldn't access any private information without the password. However, if the attacker steals the device when it's unlocked, the defense mechanism is useless, and the attacker can do whatever he or she wishes with the device. Android has a simple screen-lock pattern mechanism, but normal applications can't override the "Home" button. So, any implementation or enhancement to the login mechanism requires modifying the system. We classify the effort needed to develop such a solution as low.

Firewall. Firewalls can protect against information leaks through any network interface. Using either stateless or stateful content inspection on the communication medium, a firewall can decide whether an application is sending sensitive or confidential information to determine whether to block communication. Because the firewall operates on the kernel's lowest levels, malicious applications can't bypass it (absent exploitable vulnerabilities in the Linux kernel or system libraries). It can also work hand-in-hand with an access control mechanism, such as SELinux, to provide greater protection. Nevertheless, network interfaces aren't the only path malware can take to leak private data from the device; an alternative attack would be to send the data through SMS/MMS messages. Unfortunately, firewalls can't block such an attack.

Data encryption. Encrypting data is an excellent way to counter private data exposure. Because only the owner knows the key for deciphering data, the information is secure even if an attacker steals the device and has full access, because he or she can't decipher the encryption in a reasonable amount of time. Encrypting sensitive data handled by core applications (such as SMS messaging, emails, and contacts) will require developers to modify those applications. We classify realizing this effort as low.

Context-aware access control. By employing context-aware access control (CAAC), users can limit access to their private data depending on the context in which the device is operating. Among the contextual factors are location, time, the cellular network, and whether the device is connected to Wi-Fi and other similar elements. Such a mechanism could defend against various information disclosure attacks depending on the surrounding circumstances. If the attack occurs while the device is in a context that allows access to information, the device will permit access and disclose information. However, if the device is stolen and transferred to a foreign location, for example, the data would be secure and inaccessible to the attacker. This solution's primary challenge is to define policies and procedures that the device would implement either automatically or via manual input. Implementing this solution would require medium effort and a system modification.

Remote management. Remote management capabilities, when combined with additional security solutions, such as a firewall or context-aware access control mechanism, can improve security substantially. If an attacker steals a device, users could protect information remotely by turning on a defensive mechanism. Even during the device's everyday operation, if the remote manager can identify a worm prowling the cellular or wireless network, he or she could configure the firewall accordingly to block the worm and prevent any information disclosure. Due to the high availability the enterprise market requires, such an implementation needs a medium effort, given that the required privileges for access to relevant information and for effectively controlling the device would mandate a system modification. Nevertheless, remotely managed protection features depend on human intervention during or prior to an attack. Moreover, to defend against attacks at the right time, a remote manager must constantly monitor the device. Such a requirement is likely to be costly in terms of device resources and might place excessive demands on the remote manager.

Threat Cluster 4

The fourth threat cluster involves attackers draining a mobile device's resources. Applications for Android have neither disk storage nor memory (RAM) quotas, and hogging the CPU is also possible. This threat cluster has two primary solutions.

Resource management. The resource management security solution mitigates the threat of malicious applications draining resources. This mechanism consists of fairly allocating resources to applications according to their needs and importance (for example, the phone application is very important and should thus receive more CPU than a game). In this case, unsupervised

Taxonomy of Threats to the Android Framework

1. Abuse of costly services and functions (for example, sending short message service/multimedia messaging service (SMS/MMS) messages, making phone calls, or redirecting phone calls to high-rate numbers) by remotely exploiting a vulnerability in a core component that's exposed on the Internet.
2. Malicious activity against a network or network device (for example, sending spam, infecting other devices, sniffing, or scanning) by remotely exploiting a vulnerability in a core component that's exposed on the Internet.
3. Abuse of costly services and functions (such as sending SMS/MMS messages, making phone calls, or redirecting phone calls to high-rate numbers) by an application exploiting a vulnerability in a core component.
4. Malicious activity against a network or network device (for example, sending spam, infecting other devices, sniffing, or scanning) by an application exploiting a vulnerability in a core component.
5. Abuse of costly services and functions (such as sending SMS/MMS messages, making phone calls, or redirecting phone calls to high-rate numbers) by maliciously using the permissions granted by the owner at installation.
6. Malicious activity against a network or network device (for example, sending spam, infecting other devices, sniffing, or scanning) by maliciously using the permissions granted by the owner at installation.
7. Disabling applications or the device by remotely exploiting a vulnerability in a core component that is exposed on the Internet.
8. Disabling applications or the device by an application exploiting a vulnerability in a core component.
9. Disabling applications or the device by maliciously using the permissions granted by the owner at installation.
10. Corrupting or modifying private content, or blocking, modifying, or eavesdropping on the device's communication network (for example, phone calls, Internet communication, emails, or SMS/MMS messages) by remotely exploiting a vulnerability in a core component that's exposed on the Internet.
11. Corrupting or modifying private content, or blocking, modifying, or eavesdropping on the device's communication network by an application exploiting a vulnerability in a core component.
12. Corrupting or modifying private content, or blocking, modifying, or eavesdropping on the device's communication network by maliciously using the permissions granted by the owner at installation.
13. Obtaining, corrupting, or modifying private content when browsing a malicious Web site.
14. Blocking, modifying, or eavesdropping on the device's communication network when connected to an unreliable network.
15. Receiving spam, SMS/MMS messages, or emails.
16. Pushing advertisements to the browser application when browsing the Internet.
17. Loss of hardware components.
18. Causing a malfunction in hardware components.

resource drainage isn't possible. If a resource management solution maintains disk storage quotas and disk and network I/O are rate limited and permitted up to a certain quota, then it can fully mitigate a DoS attack. However, implementing such resource management configurations requires system modifications. Due to such a change's invasiveness, the implementation effort ranges from medium to high, depending on the exact implementation (enabling a configuration or adjusting the framework to support such an implementation).

Intrusion-detection/prevention system. A host-based IDS can counter malicious drainage in the battery, memory, or CPU by detecting abnormal rate changes in resource levels. In practice, any malware aims to remain undetected, so the IDS should continuously maintain and validate the normal usage profile.

Threat Cluster 5

The final threat cluster deals with compromised internal or protected networks. Attackers can use Android devices to compromise other devices, computers, or networks by running network or port scanners, SMS/MMS/email worms, and various other attacks.

Virtual private network. A VPN solution relies on mature principles such as message authentication codes and encryption to protect communication. The Point-to-Point Tunneling Protocol (PPTP), Layer 2 Tunneling Protocol (L2TP), and IPSec-based VPN connections are enabled on Android release 1.6 (that is, the Donut update). Enabling additional Linux-based VPN solutions on Android requires only a low effort.

Remote management. Enforcing a security policy when dealing with internal or protected networks is easy using a centralized remote management framework controlled by network administrators. However, threat mitigation's effectiveness depends on the administrator's vigilance—that is, a human factor, which is the biggest chink in the solution's armor.

Context-aware access control. When dealing with internal or protected networks, we can view context-aware access control as an automated version of the remote management approach. Upon detecting a context involving an active connection to the internal or protected network, the CAAC mechanism can increase the active security measures on the device. Such measures might include connection encryption, authentication, and more.

All in all, our review indicates that the defensive shell around Android was designed with extensive care, as the security mechanisms incorporated in Android aim to tackle a broad range of security threats. To further harden Android devices and enable them to cope with high-risk threats, we proposed several security countermeasures. We subsequently rated and prioritized the selected countermeasures (presented in Table 3) based on the following three parameters: mitigation level; the effort required for implementation; and threat clusters that the evaluated countermeasure tackles.

Based on the countermeasures Table 3 presents, we have the following security recommendations. First, Android should incorporate a mechanism that can prevent or contain potential damage stemming from an attack on the Linux-kernel layer, such as the SELinux access control mechanism. Several vulnerabilities and bugs have already exploited these pathways to gain and maintain root permission on the device. Second, the platform needs better protection for hardening the Android permission mechanism or for detecting misuse of granted permissions. Consequently, we assign the highest priority to SELinux along with a firewall, IDS, and the CAAC solutions. We place at a lower priority data encryption, spam filtering, and selective Android permission mechanisms. We recommend the remote management, VPN, and login solutions to provide telecom operators with a competitive edge when targeting corporate customers. ☐

References

1. C. Dagon, T. Martin, and T. Starner, "Mobile Phones as Computing Devices: the Viruses Are Coming," *IEEE Pervasive Computing*, vol. 3, no. 4, 2004, pp. 11–15.
2. J. Cheng et al., "SmartSiren: Virus Detection and Alert for Smartphones," *Proc. 5th Int'l Conf. Mobile Systems, Applications and Services* (MobiSys 07), ACM Press, 2007, pp. 258–271.
3. A. Gostev, "Mobile Malware Evolution: An Overview," Viruslist.com, 2006; www.viruslist.com/en/analysis?pubid=200119916.
4. D. Muthukumaran et al., "Measuring Integrity on Mobile Phone Systems," *Proc. 13th ACM Symp. Access Control Models and Technologies*, ACM Press, 2008, pp. 155–164.
5. D. Emm, "Mobile Malware—New Avenues," *Network Security*, vol. 2006, no. 11, 2006, pp. 4–6.
6. E.E. Schultz, "Where Have the Worms and Viruses Gone? New Trends in Malware," *Computer Fraud and Security*, vol. 2006, no. 7, 2006, pp. 4–8.
7. N. Leavitt, "Mobile Phones: The Next Frontier for Hackers?" *Computer*, vol. 38, no. 4, 2005, pp. 20–23.
8. M. Pelino, *Predictions 2010: Enterprise Mobility Accelerates Again*, Forrester, 2009.
9. W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security & Privacy*, vol. 7, no. 1, 2009, pp. 50–57.
10. A. Shabtai, Y. Fledel, and Y. Elovici, "Securing Android-Powered Mobile Devices Using SELinux," *IEEE Security & Privacy*, to appear, 2010.

Asaf Shabtai, *CISSP, is a PhD student at Ben-Gurion University of the Negev (BGU), Israel, and an R&D manager at Deutsche Telekom Labs. His main areas of interests are computer and network security, data mining, temporal data mining, and temporal reasoning. Shabtai has an MSc in information systems engineering from BGU. Contact him at shabtaia@bgu.ac.il.*

Yuval Fledel *is an MSc student in the information systems engineering department at Ben-Gurion University of the Negev (BGU), Israel, and a senior system architect and security expert at Deutsche Telekom Labs. His primary areas of interest are computer and network security, operating systems security, general hacking, and reverse engineering. Fledel has a BSc in software engineering from BGU. Contact him at fledely@bgu.ac.il.*

Uri Kanonov *has a BSc in software engineering from Ben-Gurion University of the Negev, Israel. Contact him at kanonov@bgu.ac.il.*

Yuval Elovici *is the director of Deutsche Telecom Laboratories at Ben-Gurion University of the Negev (BGU), Israel, and is a member of the information systems engineering department. His main areas of interest are computer and network security, information retrieval, and data mining. Elovici has a PhD in information systems from Tel-Aviv University, Israel. Contact him at elovici@bgu.ac.il.*

Shlomi Dolev *is a professor in the Department of Computer Science at Ben-Gurion University of the Negev, Israel. His recent interests include cryptography, security, and optical computing. Dolev has an MSc and DSc in computer science from the Technion, Israel Institute of Technology. He's the author of* Self-Stabilization *(MIT Press, 2000). Contact him at dolev@bgu.ac.il.*

Chanan Glezer *is a research associate at Deutsche Telekom Laboratories. His main areas of interest are organizational computing, electronic commerce, and IT security. Glezer has an MBA with a specialization in information systems from Tel Aviv University and a PhD in management information systems from Texas Tech University. Contact him at chanan@bgu.ac.il.*