A Local Cross-Site Scripting Attack against Android Phones

Michael Backes[*][†], Sebastian Gerling[*], Philipp von Styp-Rekowsky[*]

[*] *Saarland University, Germany*
[†] *MPI-SWS*

`http://www.infsec.cs.uni-saarland.de/projects/android-vuln/`

Abstract

In the first quarter of 2011, Android has become the top-selling operating system for smartphones. Its increase in market share has made Google's mobile operating system an attractive target for attackers. In this paper, we present a novel cross-site scripting attack that allows unprompted installation of arbitrary applications from the Android Market. Our attack is based on a single malicious application, which, in contrast to previously known attacks, does not require the user to grant it any permissions.

1 Introduction

In the last three years, smartphones have become our daily companions not only for business purposes, but also in our private lives, providing assistance and comfortable usage in a variety of situations. Besides the obvious capability of smartphones to make phone calls and to organize our contacts and calendar, they are usually also able to record audio and video and to track our position using GPS. They can be used for reading our e-mails, for accessing various kinds of web services (e.g. social networks), and serve as storage for privacy sensitive information. If an attacker gained remote access to our smartphone, he would be able to monitor all of our daily activities.

The attack we present in this paper targets Google's newest Android version 2.3.4 (can be run also on the Android Tablet version 3.01), which was released in April 2011. However, we expect that the attack can also be mounted on all previous versions of Android. We developed a proof-of-concept application that, without requiring any permissions, is capable of:

- Stealing cookies stored in the web browser for Web sites of the attacker's choice.
- Automatically installing arbitrary applications from the Android Market without user consent.

1.1 Paper Outline

Section 3 and Section 4 provide background information on the Android operating system and our attacker model. In Section 5 we present our attack and provide implementation details. After discussing related work in Section 6, we conclude with Section 7.

2 Responsible Disclosure

We informed Google about this weakness on June 20th, 2011. They confirmed the bug on June 21st, 2011 and started to work on a fix. The bug has been assigned the reference number CVE-2011-2357. Google notified us on July 23rd, 2011 that the problem has been fixed in Android 3.2 which will be shipped e.g. to Motorola Xoom users as build number HTJ85B. Further, they informed us on July 26th, 2011 that for smartphones the code base has been fixed as well and that the patch will be shipped in the future Android version 2.3.5. Hopefully this version will also be available as an update for current phones soon. In order to give Google the chance to fix this vulnerability we kept its details secret and only published the hash value and a short overview on our homepage and on `arxiv.org` [2].

3 Android Primer

The Android operating system is based on a Linux kernel with the Android software stack on top. This stack includes native libraries (e.g. SSL, WebKit, OpenGL) as well as an application framework and a register-based virtual machine called Dalvik, which serves as the runtime environment for user-space applications [1]. It usually runs on hardware based on the ARM architecture.

All applications installed on the phone run inside their own instance of the Dalvik virtual machine. Each application is assigned a separate Linux user id and has access to its own protected storage. The operating system is shipped with some default applications; the interesting application for our attack is the included WebKit-based web browser.

Applications for Android are written in Java (they can include native code written in C or C++). The Java code is first compiled into Java bytecode and afterwards into Dalvik bytecode. The Dalvik bytecode is stored in Dalvik executable files (.dex files) and then bundled in an Android application package (.apk file). Depending on which OS functionalities an application wants to access, the developer has to define the permissions his application requires. All permissions an applications needs are specified in the AndroidManifest.xml file, which is part of the application package. Typical permissions comprise for example the *android.permission.INTERNET* permission, which allows applications to open network connections.

The permissions that were defined by the developer are shown to the user during the installation process of the application. Upon seeing them, the user can either decide to install the application and grant it all requested permissions, or he can abort the installation process. Permissions can neither be granted individually nor can they be granted or revoked at runtime. The user either accepts all requested permissions, or he cannot install the application [1].

Finally, Android provides a mechanism that allows communication between any application component installed on the device. This communication mechanism is based on the asynchronous exchange of messages, called *Intents*, which can be either *explicit* or *implicit*. Explicit Intents explicitly specify the Intent receiver, e.g. the Java class which should be called. Implicit Intents ask the system to perform a particular service without declaring a specific receiver. These implicit Intent messages contain an abstract description of an *action* to be performed and optionally an URI that specifies the data on

```
Uri uri = Uri.parse(”https://market.android.com/”);
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);

Thread.sleep(10000);

uri = Uri.parse(”javascript:alert(document.cookie);”);
intent.setData(uri);
startActivity(intent);
```

Figure 1: Example code for mounting the attack

which this action should operate on. For example, an application could ask the system to display a Web site by sending an Intent with action `VIEW` and an `http:` or `https:` URI. Android will then try to find a suitable Intent receiver, in this case typically a web browser, which is registered to handle `VIEW/http:` Intents. If there are several matching Intent receivers present on the device, Android will ask the user which application to use.

4 Attacker model

In order to mount our attack, we have three main requirements:

- We need the user to install our malicious application. Past attacks have shown [10] that this can in general be accomplished by integrating the malicious code into an unsuspicious application, e.g. a small game. However, a considerate user might refuse to install the application if it requires special android permissions (e.g. *android.permission.INTERNET*). We prevent the user from suspecting our application by not requiring any permissions. The only requirement is that the user installs an application that looks trustworthy.
- The user needs to store login cookies in the browser, which is similar to normal cross-site request forgery attacks. In case we want to install other Android applications, it is further required that the user is already logged in to his Google account (the one he paired his phone with). Note that due to Google's single sign-on approach, it suffices that the user is logged in to an arbitrary Google service.
- The WebKit-based browser shipped with Android needs to be the default receiver for `http:/https:` and `javascript: VIEW` Intents.

5 The Attack

Our attack exploits a flaw in the Intent handling mechanism of the Android browser. Naturally, this browser is configured to receive `VIEW` Intents that operate on `http:` and `https:` URIs. Furthermore, the browser will also handle `VIEW` Intents for `javascript:` URIs. Whenever the browser receives an Intent to view an `http:/https:` URI, it will open a new browser window and load the given Web site. However, upon reception of a view Intent for a `javascript:` URI, the browser will not open a new window but reuse the currently active window. Consequently, the javascript code given in the Intent URI will be executed in the context of the Web site that is currently loaded in the active window. This leads to a generic local XSS vulnerability, which works with arbitrary web sites.

5.1 Implementation

A malicious application can use this flaw to execute arbitrary javascript code in the context of a Web site of the attacker's choice. Example code is given in Figure 1. The application first creates a `VIEW` Intent for the target Web site, in our example `https://market.android.com` and dispatches it to the system. This will cause Android to launch the default browser and load the specified Web site. The application then sleeps for some time (in our case 10 seconds) to wait for the Web site to finish loading. It then dispatches an Intent containing javascript code to be executed within the context of the previously loaded Web site. In our example, this Intent will cause the browser to display the cookie information stored for `https://market.android.com`.

5.2 Impact

The impact of this vulnerability is critical. First of all, `VIEW` Intents for `http:`, `https:` and `javascript:` URIs are not subject to any permission checks, thus they are processed even when the dispatching application does not have any permissions. In particular, the *android.permission.INTERNET* is not required for our attack. Using appropriate javascript code, an attacker can steal stored login cookies and session ids for Web sites of his choice. Even worse, using the Web site `https://market.android.com`, the attacker can install arbitrary applications with arbitrary permissions from the Android Market without any user interaction. Even though the Android Market uses a secret token to defend against CSRF attacks, an attacker can easily construct a javascript payload that reads that token and subsequently sends valid application installation requests to the Android Market server.

5.3 Difficulty

Successful exploitation of this vulnerability is trivial. An attacker would require little knowledge to write working exploit code.

6 Related Work

With Android's increasing popularity and the first successful attacks, security researchers started to take a closer look at Android. Current research regarding the Android OS can be separated into two main areas: the Android OS itself and the third-party applications end-users can install on their phones.

In November 2010, Jon Oberheide [10] discovered an Android vulnerability that enabled a malicious application with the Internet permission to install arbitrary applications from the Android Market without user consent. To this end, his application impersonated the legit Android Market application by sending valid installation requests to the Android Market servers.

Later, in March 2011, he found a stored XSS vulnerability in the Android Market Web site. This weakness allowed for unprompted installation of arbitrary applications by tricking the user into clicking a malicious link [9].

Davi et al. showed that it is possible to mount privilege escalation attacks on Android [4]. Applications are able to communicate with each other via Android's messaging system described in section 3. If we consider the situation that one application is installed without any permissions, and a second application is installed with more permissions, the following may happen: Either intentionally or due to an implementation fault, it is possible that the first application gains access to functionality that only the second application is allowed to access.

Schlegel et al. presented a concept for a context-aware Trojan that tries to efficiently hide on the mobile phone while collecting sensitive data [4]. In order to send all information the Trojan collected to a third-party server, they suggest to either make use of the previously described privilege escalation approach by Davi et al., or to use the weakness presented in the talk by Lineberry et al. [8] that relates to opening a Web site in the web browser without having the Internet permission. The latter is also used in our approach (cf. Figure 1).

Since the Android Market does not implement a real review process of uploaded applications (as it exists for the Apple App Store), the decision whether an application is trustworthy and secure is mainly up to the user. The only barrier for publishing applications is to pay 25$ for creating a developer account. Therefore, the second line of research focuses on the security of third-party applications.

SCanDroid reasons about the security of applications by analyzing data-flows in an offline setting [3], whereas Taintdroid follows a real-time approach by conducting a system-wide dynamic taint tracking and analysis [5]. Enck et al. introduce a system called Kirin that aims at preventing the installation of applications with critical permission combinations (e.g. microphone and Internet access in conjunction could resemble a wiretap application) [7]. The results of their recent analysis of 1.100 popular Android applications show that it is necessary to look at privacy and security issues of applications [6].

7 Conclusion

Our attack poses a severe threat to all currently deployed Android systems. It opens up a lot of new opportunities to compromise core functionalities of Android phones and thus constitutes a good starting point for further attacks. Nonetheless, the current approach already suffices to steal stored browser cookies and to install other Android applications without user interaction.

References

[1] Android developer's guide. Available from: `http://developer.android.com/guide/index.html`.

[2] Michael Backes, Sebastian Gerling, and Philipp von Styp-Rekowsky. A novel attack against android phones. Available from: `http://arxiv.org/abs/1106.4184`.

[3] Avik Chaudhuri, Adam Fuchs, and Jeffrey Foster. SCanDroid: Automated security certification of android applications. 2009.

[4] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Information Security: 13th International Conference (ISC 2010)*, pages 346–360, 2010.

[5] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, pages 393–407, 2010.

[6] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proc. 20th Usenix Security Symposium*, 2011.

[7] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. On lightweight mobile phone application certification. In *Proc. 16th ACM Conference on Computer and Communication Security (CCS 2009)*, pages 235–245, 2009.

[8] Anthony Lineberry, David Luke Rechardson, and Tim Wyatt. These aren't the permissions you're looking for. Available from: `http://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf`.

[9] John Oberheide. How i almost won pwn2own via xss. Available from: `http://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/`.

[10] John Oberheide. When angry birds attack: Android edition. Available from: `http://jon.oberheide.org/blog/2011/05/28/when-angry-birds-attack-android-edition/`.