Technical Report

Nr. TUD-CS-2015-0065

April 1st, 2015

An Investigation of the Android/BadAccents Malware which Exploits a new Android Tapjacking Attack

TECHNISCHE UNIVERSITÄT DARMSTADT

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN

Authors
Siegfried Rasthofer[1]
Irfan Asrar[3]
Stephan Huber[2]
Eric Bodden[1,2]

[1]EC SPRIDE / Technische Universität Darmstadt
[2]Fraunhofer SIT
[3]Appthority

An Investigation of the Android/BadAccents Malware which Exploits a new Android Tapjacking Attack

Siegfried Rasthofer[1], Irfan Asrar[3], Stephan Huber[2], Eric Bodden[1,2]

Center for Advanced Security Research Darmstadt (CASED)
[1] Technische Universität Darmstadt, Germany
siegfried.rasthofer@cased.de
[2] Fraunhofer SIT, Darmstadt, Germany
{stephan.huber, eric.bodden}@sit.fraunhofer.de
[3] Appthority, USA
iasrar@appthority.com

Abstract. We report on a new threat campaign, underway in Korea, which infected around 20,000 Android users within two months. The campaign attacked mobile users with malicious applications spread via different channels, such as email attachments or SMS spam. A detailed investigation of the Android malware resulted in the identification of a new Android malware family Android/BadAccents. The family represents current state-of-the-art in mobile malware development for banking trojans.
In this paper, we describe in detail the techniques this malware family uses and confront them with current state-of-the-art static and dynamic code-analysis techniques for Android applications. We highlight various challenges for automatic malware analysis frameworks that significantly hinder the fully automatic detection of malicious components in the malware. Furthermore, the malware exploits a previously unknown *tapjacking* vulnerability in the Android operating system, which we describe in detail. As a result of this work, the vulnerability, affecting all Android versions, has been patched in the Android Open Source Project.

Keywords: Botnet, Threat Campaign, Android Malware, Code Analysis, Banking Trojans, Vulnerability

1 Introduction

According to a recent study [9], Android has reached a mobile market share of 81%. There is an app for almost every need provided by various app stores such as the Google PlayStore with 1.3M applications by July 2014 [33]. Beside apps that are used mostly for amusement, there are also more critical applications that handle confidential data such as mobile banking applications. According to a Federal Reserve Board study [25], more and more people switch from manual usage of an ATM to a more convenient way of using mobile banking with their smartphones. This is a very attractive target for attackers who want to steal

money from victims. Indeed, there is a big underground market for trading stolen bank account credentials [34]. For instance, Symantec reported [34] that a single underground group made $4.3 million in purchases using stolen credit cards over a two-year period.

The Android operating system got enhanced with different security features, such as the 'Application verification' in version 4.2. Its goal is to protect the user against harmful applications. Despite those protection mechanisms, banking trojans are still actively spread and successfully exploited in many applications [6]; even worse McAfee [19] is predicting a rapid growth. Very recently, we identified a new threat campaign underway in South Korea which emphasizes McAfee's prediction. The campaign stole, within two months, more than 20,000 bank account credentials of users mostly resident in Korea. We identified a new malware family `Android/BadAccents` (named after the main component in the first stage of the trojan) that impersonates known banking applications in order to steal the user's credentials. Furthermore, it also steals incoming SMS messages, aborts phone calls and installs a fake anti-virus application.

The goals and contributions of this paper are three-fold. First, we describe in detail the techniques this malware family uses, and describe the current state-of-the-art of banking trojans. Second, we confront these techniques with current state-of-the-art code analysis techniques for Android applications. We describe challenges that static as well as dynamic code analysis techniques have to cope with in order to automatically analyze current malware. For instance, hiding sensitive information in native code is no longer a theoretical problem for static analysis, it is already exploited in the wild. The usage of multi-stage command and control (C&C) protocols is growing into a challenge for dynamic code-analysis techniques as well. Even malware analysis frameworks that try to circumvent emulator-detection mechanisms [26] are not well prepared for current Android malware. There is still a big need for a proper environment setup, such as specific files on the SD card or specific apps installed, as otherwise the malicious behavior does not get triggered. These are significant challenges that the future state-of-the-art code-analysis approaches will need to address. Finally, we report on a dangerous `tapjacking attack` on Android that we have identified and which has been exploited by the malware family. Tapjacking, which is similar to clickjacking for web applications, is an attack where the user clicks/-taps on seemingly benign objects in applications, triggering actions not actually intended by the victim [23]. This results in dangerous security issues as we will elaborate in our paper. The attack seems to apply to all currently available Android versions. The attack, together with a patch, has already been submitted to the Android Security team who confirmed our attack and updated the latest Android version.

The rest of the paper is organized as follows. Section 2 describes the identification of the malware and Section 3 describes the details of the malware. In Section 4 we identify the challenges for current state-of-the-art code analysis techniques. The new AOSP vulnerability together with the mitigation technique

is shown in Section 5. Section 7 describes the related work in the field of Android security while Section 8 concludes the paper.

2 Threat Identification

Mobile malware that is targeting Korea in many ways represents the best of breed practices when it comes to mobile malware development. Excluding spyware, the majority of threat families targeting mobile devices in Korea involve a C&C component. They employ tactics such as social engineering, apply themes from trending events such as movies to famous brands such as StarBucks and usually target high value assets like bank accounts. The same applies to Android/BadAccents. Shortly after the news broke that the movie 'The Interview', originally scheduled to be released on Christmas Day, would appear online from Sony Pictures, numerous sites claimed to offer a pirated copy fueled by the rumors that the movie might be distributed online for free due to the circumstances surrounding the change of film distribution channels. One free copy making the rounds in South Korea turned out to be an Android Trojan that we have designated Android/BadAccents (named after the main component in the first stage of the Trojan).

Even though Google Play was used in the past as a distribution vector for Korean malware, improvements in the Google Play screening process have pushed the malware authors to use other distribution vectors. The enforcement of strict piracy laws in Korea means that third party sites for Android app downloads are not very common. Thus distribution vectors such as file torrents, spoofed emails pretending to be updates from a bank, as well as sms spam tend to be the most common distribution methods. As if the complexities called out above were not enough, the sheer volume of the files, spam messages or artifacts of interest involved in typical campaigns that can span from a few days to months, leaves little room for manual malware analysis framework.

However, with the help of a threat-detection framework that uses standard practices in the AV industry, we were able to spot out an interesting malware campaign underway in Korea. The framework is designed as a multi-stage process. It continuously crawls the web to acquire applications from file repositories, file sharing sites or specific app markets. The framework first checks the applications for known malware to identify threat families. If none were found, but the context of the samples still indicate suspiciousness, the system involves a further investigation. Suspicious applications are identified with various static as well as dynamic analysis approaches. Since the framework itself is a commercial and internal detection framework, we cannot provide concrete details about the detection mechanism, since attackers could use this information for re-designing their malware development process. This would threaten further malware analysis. Therefore, we will give only a brief overview about the framework in the following.

As a first step, all crawled applications are inspected with static analysis techniques to search for previously seen patterns. Several strategies can be em-

ployed here to identify samples that could potentially be malicious. The strategy includes but not limited to

- Identify samples with permissions that tend to be suspicious.
- Identify strings in samples that can be correlated to external data feeds, such as SMS numbers associated with malware, or urls associated with C&C activity.
- Samples that have been signed with certificates that have been previously seen in malicious campaigns.
- Samples that have been associated with spam distribution campaigns.

Once these samples of interest have been identified via static analysis (as described above) the next step is to further filter the sample set by dynamic analysis in a proper sandbox. Similar solutions from academic research are Andrubis [20] and MobileSandbox [32]. The dynamic analysis adds additional data to the investigation. For instance, it includes a traffic analysis of sms sent/received as well as ip address of the destinations being reached out to in addition to file system monitoring.

The framework identified the Android/BadAccents malware since many indicators pointed to a suspicious banking application. For example, constant strings stored inside a file indicating a banking application. Additionally, strings such as 'The Financial Supervisory Authority' or 'electronic fraud prevention service' are usually an indicator for message dialogs used in social engineering attacks. The framework contains some heuristics that identified an email component inside the application, probably used for leaking sensitive user data, but no concrete information about the username and password of the account were available. Therefore, we needed a further manual analysis which we will describe in the following section.

3 Android/BadAccent Malware

The results described in the previous section gave us an indication about the malware family (banking trojan), and evidence about some data being sent via email. Important questions such as *"What kind of sensitive data gets leaked?"*, *"Are sensitive data leaked via email?"* or *"What is the username and password of the target email account?"* could not be answered yet at this stage. However, these are very important questions for security analysts in the case of active malware, because the analysts have to initiate further steps to remove the threat, for instance a C&C server takedown (see section 6). Since the questions above relate to dataflow tracking problems [4], we used current state-of-the-art static (e.g., FlowDroid [4]) as well as dynamic (e.g., TaintDroid [11]) dataflow-analysis tools to investigate further. Unfortunately, none of the tools could answer these questions and we elaborate in this paper why this was the case and what are the challenges for code analysis tools in analyzing state-of-the-art malware on the example of Android/BadAccents (section 4). We therefore manual reverse

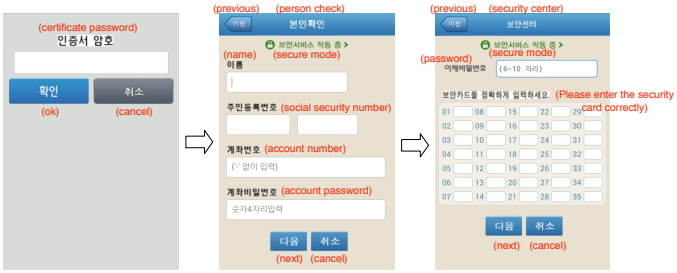

Fig. 1. Confidential banking credentials that get stolen

engineered the malware with the help of `CodeInspect` [27], an interactive debugger for Android bytecode. It relies on the Soot [1] framework and its Jimple intermediate representation [3].

In general, Android/BadAccents is a banking trojan that tries to steal bank account credentials through a phishing attack. The victim is asked to enter her confidential data into a Graphical User Interface (GUI) that looks identical to the one of a benign mobile banking application. But the malware's GUI is designed by the attacker, and is instrumented to steal the credentials. Figure 1 shows such a fake GUI component which appears after a fake security message which prompts the user for some action.

Android/BadAccents demonstrates the complexity of current Android banking trojans. Different interactions, environment settings and conditions are necessary before a specific malicious behavior gets triggered. The malware sample uses different techniques to hide the malicious behavior as long as possible. Figure 2 gives an overview of the main components in the Android/BadAccents malware and shows the complexity of environment settings, workflow and external events that are involved. Especially the *Intercept SMS* components shows, that current attackers do not only rely on a single channel for transmitting sensitive data. Instead they use several ones, in this case e-mail and the HTTP connections. In the following we will describe each component individually in detail, and its requirements for triggering a malicious behavior. The requirements on code analysis tools are evaluated afterwards.

3.1 Send SMS

The *Send SMS* component gets activated at application startup time and is responsible for sending SMS messages to all contacts on the phone that have more than 5 digits as a phone number. It first initializes a connection to the C&C server (the victim's device phone number is used for authentication) from which it receives the text for the SMS message. Additionally, it saves all phone numbers of the contacts into a global storage (SharedPreferences file). After receiving the text, the component immediately sends a message containing that text to all contacts. This mechanism is probably used for spreading the malware

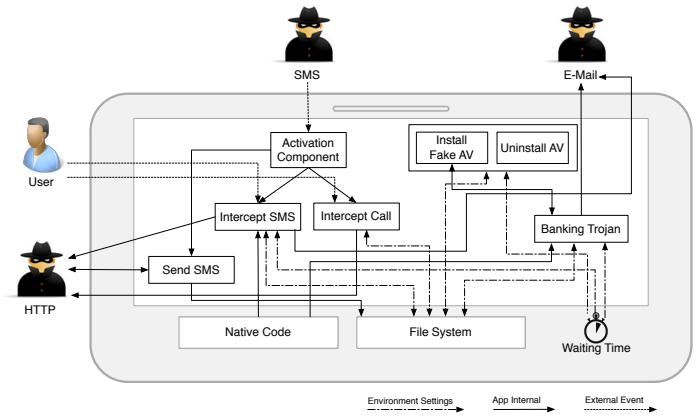

Fig. 2. Interactions and environment settings necessary for triggering malicious behavior in the Android/BadAccents Malware

to all contacts. We assume that the text from the C&C server contains spam messages together with a download link to the Android/BadAccents malware. The attacker's aim is to infect the SMS receiver with additional malware by clicking on the link.

Pre-Condition1:
- There has to be at least one contact on the device
- The phone number of the contact has to have more than 5 digits

Pre-Condition2:
- There has to be an internet connection to the C&C server

Post-Condition1:
- Phone numbers of all contacts are stored in global share (SharedPreferences file)

3.2 Activation Component

The *Activation Component* is responsible for receiving C&C messages via SMS. Using SMS as a protocol is an important design decision that is different from traditional IP-based approaches known from infected PCs. Zeng et al. [39] already illustrated this design in 2012. The main advantages of an SMS-based approach instead of IP-based one are the fact that it does not require steady connections, that SMS is ubiquitous, and that SMS can accommodate offline bots easily.

The *Activation Component* is implemented as a broadcast receiver, which is active from the time the application starts. This broadcast receiver registers 63 different actions it can react on. However, it uses only a single one of them, the receiving SMS action. It intercepts all incoming SMS messages and triggers

the malicious behavior only if the message contains special commands. More concretely, it is responsible for activating the *Intercept SMS* and *Intercept Call* component (details below). The Android/BadAccents malware contains two specific checks on the incoming SMS number. It checks for '+84' and '+82' numbers, which indicates that the malware expects SMS from a C&C SMS server either located in China or South Korea. The message has to have a special format that contains either 'sd_$\langle MESSAGE \rangle$', 'ak40_0', 'ak40_1' 'call_0' or 'call_1' and can be concatenated with '_' (e.g., 'ak40_1_call_0'). The 'ak40' command is responsible for the *Intercept SMS* component and activates that component with 'ak40_1' and deactivates it with 'ak40_0'. The 'call' command is responsible for the *Intercept Call* component and 'call_1' activates and 'call_0' deactivates it. Activating a component is implemented by storing activation-flags (e.g. $\langle call, 1 \rangle$) into a SharedPreferences file, deactivating components is done by storing deactivation-flags (e.g. $\langle call, 0 \rangle$). The individual components get called in a specific time interval were they first check for the appropriate activation-flag before running it. This is indicated as dotted arrows in Figure 2 from both components to *File System*. The 'sd_$\langle MESSAGE \rangle$' command is equivalent to the functionality of the *Send SMS* component (see section 3.1). The main difference is the communication channel. Instead of receiving the text of the message body via HTTP (*Send SMS* component), it uses only the SMS channel by taking the message from the incoming C&C SMS ($\langle MESSAGE \rangle$) .

Pre-Condition3:
- send an SMS containing $sd\_\langle MESSAGE \rangle$ to the mobile device
- *Pre-Condition1* has to be satisfied

Pre-Condition4:
- sending an SMS containing $ak40\_1$ to the mobile device

Pre-Condition5:
- sending an SMS containing $call\_1$ to the mobile device

Intercept SMS This component intercepts all incoming SMS messages that do not contain any C&C command and forwards them to the attacker via HTTP and E-mail. It uses two channels in parallel for a more reliable data theft. The credentials of the E-mail account are hidden in native code which makes the detection hard for static analysis approaches that operate purely on the Dalvik bytecode. Listing 1.1 shows two native methods that return the constant username and password (original credentials are removed) that get called in the `onCreate` method (listing 1.2) and stored into a SharedPreferences file (`setValue` method). Before sending the email, the credentials are extracted from the SharedPreferences file in order to authenticate against the email server.

```
1 void Java_com_MainActivity_stringUser() {
2   return "USERNAME";
3 }
4
5 void Java_com_MainActivity_stringPassword() {
6   return "PASSWORD";
7 }
```

Listing 1.1. Methods in Native Code

```
1  public native java.lang.String
       stringPassword();
2  public native java.lang.String stringUser();
3
4  public void onCreate(Bundle b) {
5  ...
6  user = stringUser();
7  setValue("musername", user);
8  pw = stringPassword();
9  setValue("mpass", pw);
10 ...
11 }
```

Listing 1.2. Accessing Native Methods within Java

Pre-Condition6:
- Wait for 20 seconds
- Pre-Condition4 has to be satisfied ('ak40_1' has to be sent)
- There has to be an internet connection

Intercept Call The *Intercept Call* component intercepts all incoming calls and checks whether the caller is stored as a contact on the device or not. If this is not the case, the call gets aborted and the entry in the call log gets deleted. We assume that the attackers want to abort calls from the bank which could have detected suspicious transactions caused by the banking trojan.

Pre-Condition7:
- Pre-Condition1 has to be satisfied
- Incoming-call number does not have to be part of the list in Post-Condition1 has to be satisfied
- Pre-Condition5 has to be satisfied ('call_1' has to be sent)

3.3 Install/Uninstall

The Install/Uninstall component first removes one particular app, the 'AhnLab V3 Mobile Plus 2.0'[1] app in case it is installed on the device. This is a malware-scanner application especially designed for detecting banking trojans. In the *Banking Trojan* component, a fake 'AhnLab V3 Mobile Plus 2.0' application gets installed which impersonates the original app and which contains malicious components similar to Android/BadAccents.

Pre-Condition8:
- Wait for 40 seconds
- 'AhnLab V3 Mobile Plus 2.0' app has to be installed

3.4 Banking Trojan

The main component of Android/BadAccents is the *Banking Trojan* component. As a first step, it tries to hide the application's icon from the launcher.

[1] `https://play.google.com/store/apps/details?id=com.ahnlab.v3mobileplus`

This is possible with a singe API call (*setComponentEnabledSetting* in *PackageManager*) and does not require any permission. After a delay of 30 minutes, the malware looks for DER-formatted certificates stored under a specific folder on the SD card. If that is the case, the malware checks whether the user has installed specific korean banking applications such as Shinhan Bank, Woori Bank or NH Bank. This indicates that the threat campaign primary targets user from Korea. Next, if one of these application is installed, it dynamically creates a new view impersonating this app. The 'fake' app uses social engineering in showing security warnings that should convince the user to provide the attacker her data.

After accepting the security messages, the attacker tries to steal the banking credentials of the victim. Figure 1 shows the individual GUI fields the user has to go through. It is worth mentioning that input into the fields has to satisfy specific criteria such as the certificate password has to be entered twice or the password in the *security center* has to have more than 5 digits. If everything got filled out correctly, all the data, together with the certificate gets sent to the malicious e-mail account. Similar to the *Intercept SMS* component (see section 3.2), the e-mail account credentials were taken from the native code methods.

Pre-Condition9
- 30 minutes waiting time
- DER-formatted certificate stored at specific folder on sd-card
- At least one specific Korean banking app has to be installed
- Accepting fake security warnings by clicking
- Fields correctly filled

4 Code Analysis Challenges

In the previous section, we demonstrated the different requirements that have to be met for triggering malicious behavior. In this section, we elaborate more on the challenges that apps such as the one examined here pose to current state-of-the-art malware analyses techniques. One particularly important piece of information for most practical malware investigations is the kind of communication channels the app uses. In the Android/BadAccents example, it is important to know the username and password for the email account, the C&C URLs and the number of the C&C SMS sender, since sensitive banking credentials get sent through these channels. Very similar questions arise in analyses of different malware families, since often the protection of the user can most easily be achieved by taking down the attacker's communication channels. Answering these questions in an automatic way would save a lot of time and money for the investigation. There are two code-analyses approaches that can be used: static or dynamic analysis techniques, or more likely a combination of both. Unfortunately, both approaches have some known limitations that causes problems for current malware analysis approaches. In the following we look more concretely into the different challenges for static and dynamic analysis approaches that will arise during an analysis of the Android/BadAccents example.

4.1 Static Analysis Challenges

In general, static analysis is a very powerful technique since one can reason about all paths in the application, but in the case of Android malware analysis there is no *complete view* on the code. This has various reasons which will be described using the Android/BadAccents example.

Recall that we are interested in the channels the attacker is using and we know from our investigation (section 3) that sensitive data are sent via e-mail. By answering the question *What is the username and password of the email account?*, one is interested in concrete values at specific API calls (e.g. API used for sending email). In some situations, the information itself might be stored not as part of the app but on a remote server. In this case, the information is out of reach for a static analysis. But even if it is encoded as part of the application, static analysis either backward [14] or forward [4] may still fail to extract it. One concrete example for why this might be the case is the hiding of sensitive information in *native code* within Android/BadAccents. The extraction of values encoded that way would require sophisticated analysis support for both Android bytecode and ARM native code. And even if the analysis of native code were possible, this code might load the credentials from external storage such as a SharedPreferences file. This means that the static native-code analysis would have to be able to interpret this code to be able to extract the right information from that location. All in all, the implementation burden on the analysis developer would be humongous. Furthermore, if the malware sample would include binary packers or crypters [30] additional challenges would be added to the static analysis.

As a summary, static analysis is very useful in general, but its known limitations pose a serious problem when analyzing current Android malware. Dynamic analysis promises to alleviate some of these problems.

4.2 Dynamic Analysis Challenges

For all the above reasons, dynamic analysis or *behavior analysis* [30] has been advocated in the context of malware analysis [20, 32]. To be complete however, dynamic analysis requires a set of execution traces that are representative of all the possible program behaviors. While observing all the program behaviors of a complex program is impractical, several coverage criteria have been proposed in the software testing literature to approximate full behavior coverage, their effectiveness however is still debated [16, 17]. Different facts in Android significantly hinder the triggering of malicious behavior by dynamically executing the code. For the example of Android/BadAccents we summarize the major problems in the following three categories: external events, environment settings and user interaction.

External Events The Android OS is an event-driven environment that reacts to various events and executes the registered event handlers. For instance, an incoming phone call is modeled as an Android internal event, called intent [37],

that can be intercepted through a corresponding callback defined in the application. This produces the first challenge: a simple dynamic analysis is insufficient if it fails to generate the proper events. Researchers have proposed several approaches [31] for fuzzy testing Android components by sending abnormal/random intents to the components in order to identify security bugs. Nevertheless, section 3.2 shows that the malicious behavior gets only triggered if, for instance, the incoming SMS or HTTP request have the proper format. Furthermore, the ordering of events can also matter. For instance, the *Intercept SMS* component described in section 3.2 gets only activated if the attacker first sends an 'activation-command' and second the user sends an SMS to the victim. This makes a fully automated triggering of the original *Intercept SMS* component extremely difficult.

Environment Settings A successful analysis of Android malware with a *behavior analysis* requires a properly setup environment, since many malware families check for clues of an emulated environment before they trigger their malicious behavior. The environment thus must be set up in such a way that it *emulates* all aspects of a proper smartphone. To some extent, this is impossible. For instance, emulators will always expose timing and cache behavior that is clearly distinguishable from real phones [26]. But not only emulator checks complicate dynamic analysis. The problem of *timing bombs*, were the malware waits for a specific time until it triggers its malicious behavior (see pre-requirements in section 3.2) poses a serious problem to dynamic analyses. As the Wall Street Journal reported this year[2], Android malware went undetected in the Google Play store, infecting close to 10 million devices, due to such a time bomb. Time bombs can be 'evaded' by speeding up the time in the environment. Unfortunately, this might still be insufficient with state-of-the-art malware samples. Android/BadAccents requires specific files in the file system (DER-formatted files), specific contact data stored on the device and specific apps installed on the device (Korean banking apps) before the banking trojan gets activated. Since there is an exponential amount of combinations for different settings, it is very difficult to come up with a proper setting of an environment that emulates all that.

User Interaction The functionality of most applications involve some user interactions, such as clicking on buttons, swiping objects or filling out forms. Many of these interactions may need to be emulated to facilitate a meaningful dynamic analysis. Again, there has been a lot of research in the area of Android GUI testing [2, 8] but to the best of our knowledge none of these approaches would successfully work on Android/BadAccents. For instance, the first GUI in figure 1 requires the user to input her password two times. Randomly inserting some values and automatically clicking on the 'ok'-button would not result in

[2] `http://blogs.wsj.com/personal-technology/2015/02/04/android-malware-removed-from-google-play-store-after-millions-of-downloads/`

a page switch. Also the password in the first and third screen page has to have more than 5 digits, otherwise the GUI will not switch to the next one and the malicious behavior of stealing the credential data (shown in figure 2) would not be triggered. Figuring out the right combination of inputs would require the most sophisticated techniques, such as symbolic execution, which are hard to scale in general. Further research in this field is clearly required.

5 Vulnerability and Attack Description

Among other things, the Android/BadAccents malware tries to obtain Android Device Administration privileges [35] without the user's knowledge. Once obtained, this protects the application from uninstalling it programmatically and allows it to lock the device screen without the user's agreement. When the privileges are requested, the Android OS shows a warning message to the user, who then has to accept or deny the request. The malware abuses some Android vulnerability to trick the user into accepting the administration request by a so called *tapjacking* attack. The following section describes tapjacking attack techniques, gives an overview of the Android administration API and the associated risk. Then, we show in detail the attack in Android/BadAccents and describe the vulnerability in the Android operating system. In the last subsection, we describe the patch and discuss the effectively of Android's mechanisms for protecting against tapjacking attacks.

5.1 Tapjacking Attack

The formal name or most common name in research for a tapjacking attack is UI redressing [23] and subsumes tapjacking as a specific case. The first described form of such an attack family was clickjacking in web browsers for the Adobe Flash player shown by Hansen et al. [13]. Other current attack forms include cursorjacking or filejacking. With the porting of clickjacking to the mobile context, the so-called tapjacking was born.

The basic idea behind tapjaking on Android is not to directly exploit some system vulnerabilty, instead its focus is to force the user to an interaction without her knowledge and to hide the system or application information which is shown as a consequence of this hidden interaction. A harmlessly looking overlay window is brought to the foreground, hiding the real application behind the overlay window. The design of such an overlay window can be freely defined, for instance posing as a game or some generic application dialog.

As an example imagine the user is playing a simple game tapping on different symbols located on the screen. What she doesn't know is that in the background the game application has started the phone call app and all the taps the user is performing are routed through the game directly to the dialing app. The user is thus dialing and calling some premium number without being aware because the whole procedure is hidden behind the overlay window.

The requirements for such an attack are provided by the Android UI API. Attacks can be performed in different ways. The main premise is to generate a UI element which can be layered over applications and that touch gestures on this element are routed to the underlying application. An additional requirement for successful tapjacking is the hidden start of the victim application or a part of the application [7] behind the overlay. For such hidden starts exported activities or defined `intent-filters` in applications can be used. System applications or Android settings can be accessed via system intents. To route taps through underlying applications, Android provides settings that make a widget transparent for touches. If such touch transparent windows is brought to the front of another application all touches go directly to the underlying application. As a proof of concept, Guerrero provides a tapjacking framework [10] based on Android `Toast` widget. Another approach was shown by Niemietz et al. [23] who build a transparent overlay window for logging touch gestures. Android/BadAccents uses another approach, defining a *LinearLayout* View with a specific overlay attribute.

Android Device Administration API The Android/BadAccents malware uses tapjacking to activate the Android Device Administration. To understand the Android Device Administration feature and why it can be a security risk, we first provide some background information. The Android developers introduced an API for applications to support enterprise features [35]. This *Device Administration* API provides functions on the system level, with varying security impact. It allows an app to force the user to set a device password, and can define specific password policies like the length or complexity of the password. Other feature are to enforce storage encryption, locking the device or to make a factory reset and wipe all data. The full set of supported policies can be found in the developer documentation [36]. To use the methods of the Administration API, the application must define the `android.permission.BIND_DEVICE_ADMIN` permission in the `AndroidManifest.xml` file. Additionally an XML policy file in the app declares the features which can be used [35]. These policies must cover the used API method calls in the code. After installation and after using an Administration function call in the code for the first time, the system asks the user to enable this device admin application's functionality. The request lists all declared policies and asks the user to accept or deny the request. Once this request has been accepted, the application can have access to sensitive systems and data or wipe the device. But more importantly in the case of Android/BadAccents, with the administration activation an application also earns a special kind of uninstall protection: user need to first unregister the application as an administrator under security settings options before the application can be uninstalled. To monitor the administration state of the application itself a broadcast receiver with the intent-filter `android.app.action.DEVICE_ADMIN_ENABLED` can be set in the `AndroidManifest.xml` file. With this filter the application recognizes if the device administration is accepted and activated, or disabled.

```
1 private void activateAdminReceiver() {
2     ComponentName cn = new ComponentName(this, AdminReceiver.class);
3     Intent i = new Intent("android.app.action.ADD_DEVICE_ADMIN");
4     i.putExtra("android.app.extra.DEVICE_ADMIN", cn);
5     this.startActivityForResult(i, 20);
6     createSmallWindow();
7 }
```

Listing 1.3. Isolated proof of concept code calling admin permission interface

Malware tapjacking After a detailed analyses of the malicious application, we isolated the code responsible for the tapjacking attack and reassembled it into a stand-alone proof-of-concept implementation. The malware uses the described tapjacking attack to obtain Android Device Administration privileges and thus the ability to lock the device screen. Another aspect is the uninstall protection. Once admin privileges are granted, antivirus tools cannot remove the malware any longer. The attack can be illustrated as shown in figure 3. The victim only

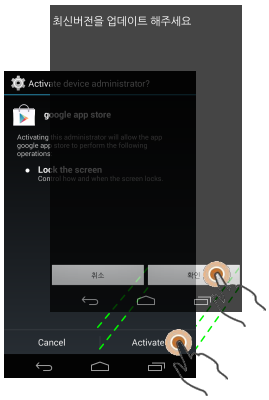

Fig. 3. Tapjacking Attack on Android Device Administrator App

sees an application window requesting *"Please update to the latest version"* with a *confirmation* and a *cancel* button. Pressing *confirmation* she activates the device administration features.

Listing 1.3 shows the first part of the tapjacking code by calling the administration request. In line 3 and 4 the device administration request intent is prepared. With the statement `startActivity` in line 5 the administration privileges request activity is started and shown to the user. The second parameter of the statement can be ignored. Due to the asynchronous execution character of Android the application does not stop after calling the administration activity. The next method call (line 6) is executed immediately providing the overlay window (see figure 3). This overlay window is an extended `LinearLayout` class defining specific layout properties. Listing 1.4 shows an excerpt of the method setting these properties. The first layout option is the overlay definition which

```
private void setupLayoutParams() {
    layoutParams = new WindowManager.LayoutParams(WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,
                WindowManager.LayoutParams.FLAG_FULLSCREEN,
               WindowManager.LayoutParams.FLAG_SCALED);
    layoutParams.flags = WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE;
    ...
}
```

Listing 1.4. Settings for ovelay window layout paramters

```
 mCancelButton = (Button) findViewById(R.id.cancel_button);
+        mCancelButton.setFilterTouchesWhenObscured(true);
         mCancelButton.setOnClickListener(new View.OnClickListener() {
...
         mActionButton = (Button) findViewById(R.id.action_button);
+        mActionButton.setFilterTouchesWhenObscured(true);
         mActionButton.setOnClickListener(new View.OnClickListener() {
```

Listing 1.5. Add tapjacking protection to DeviceAdminAdd.java file

enables the window to be displayed on top of everything else, in this case over the administration request activity. The second and third flag are responsible for the full screen size of the overlay window. The last option (`FLAG_NOT_TOUCHABLE`) is the crucial factor. It makes the window transparent for touches and therefore every touch gestures on it were received on the application behind it. Considering the malware example the victim assumes she confirms the update request, but in reality she activates the administration privileges. After enabling them the administration request is closed and the malware removes the overlay. This form of attack is working to Android Kitkat version 4.4 and older Android versions. In the newer Lollipop version (Android 5) there seems to be some conflict with the asynchronous execution, here the overlay window was always shown behind the administration request. Thus the victim would detect the attack. We slightly modified the isolated proof of concept code to show that the attack is still possible and that there is no tapjacking protection for the administration activity. We informed Google about our discovery and provided a patch preventing such an attack.

5.2 Counter-Measures

As a counter measure against *tapjacking* Android provides some specific protection mechanism. It was introduced in API level 9 (Android 2.3)and is enabled by the method `setFilterTouchesWhenObscured()`. Alternatively view elements can be protected on the level of XML declarations by defining the attribute `android:filterTouchesWhenObscured`. The code excerpt for the patch shown in listing 1.5 is setting the tapjacking protection to the accept and cancel button. The code lines marked with (+) are the contributed patch to the original code.

With this patch, if a touch event from an overlay window arrives at one of the buttons in the Administration privileges activities, the touch will be ignored. An attacker app thus can no longer trick the user into obtaining administrator privileges without her explicit consent.

Additional findings After providing the patch we additionally checked the AOSP code to see for which kinds of UI widgets the tapjacking protection is enabled for and what is the functionality behind those widgets. We found the *application installation confirmation* button, the *backup and restore allow* button, the *accessibility Service Warning dialog accepting* button and the *VPN confirmation dialog* button. This shows that only a minority of system applications use tapjacking protection. Considering other system application like telephony or SMS applications, there thus still remains some capability for tapjacking attacks. The proof of concept framework [10] demonstrates this.

Another problem is the developer awareness of tapjacking attacks against common applications. We analyzed 263.623 different apps selected from each PlayStore category (downloaded November 2014) and 1.579 open-source applications from the open-source f-droid[3] market, to see which apps have tapjacking protection enabled for at least one UI element. The result for f-droid applications was 0, while in the Google Play store we found 369 applications defining `android:filterTouchesWhenObscured` or `setFilterTouchesWhenObscured()` method. This shows that nearly all of the developers are not aware about this attack vector. The disadvantage of Android's protection concept is an insecure default. Protection must be enabled explicitly and separately for each UI element to be protected. A simple proof-of-concept tapjacking protection mechanism was developed by Niemietz et al [23]. They introduced a tapjacking security layer consisting of a transparent layer over each foreground application. If a malicious application tries to get above the victim activity to set up a tapjacking attack, the security layer can catch all the touches trying to reach the protected app. We believe that such a concept would be simpler to maintain and should be introduced.

6 Takedown

The biggest difference between a takedown - the notification of malicious content with the intention of removal to parties concerned - on Google Play and malware not hosted on Google Play is that a takedown request to Google Play tends to be pretty straight forward and after an investigation by Android Security the malicious content is usually removed within a short period of time. In the case of Android/BadAccents, it roughly took about a week to dismantle the machinery and included the involvement of several parties in different jurisdictions. The first part we contacted with Amazon Security, to inform them that the Amazon Web Service was being used to host the malicious files of the Android/BadAccents family. Despite the fact that the holidays were just around the corner, Amazon Web Service acknowledged the notification and an investigation was started. The takedown of the files was slow and it took about a week for the files to be removed and during the time consumers were still exposed to the files. Our next point of contact was the Korean Information Security Agency who immediately responded to our request and then carried out an internal investigation of their

[3] https://f-droid.org/

own. It is very likely that the so called 'Yanbian Gang' was involved in this threat campaign [15].

7 Related Work

In this section, we describe a number of related work in the context of Android malware analysis that addresses attacks and threats.

Abusing the device administration privileges in order to make the uninstallation of applications more difficult is a common technique used in Android malware. For instance, the Android malware OBAD [38] requests administration privileges. Additionally it uses an android vulnerability (fixed in Android 4) to hide its entry from the device administration list. This means it was also not possible for a user to manually revoke the admin privileges for uninstalling the malware. It also contained different environment checks for emulator detection and code obfuscations to impede manual reverse engineering. Also different ransomware applications like Android/Koler [18] try to gather administration privileges to lock the device and encrypt the data storage. Another related malware in the context of banking trojans and C&C is the Zeus [22] trojan. This banking trojan exists despite of Android also for different mobile platforms like Blackberry, Windows Mobile or Symbian. The focus of the first Zeus trojan was to steal mTAN numbers through sms interception. Newer version of Android trojans are aiming on stealing credit cards through wireless connection. Zhou et al. showed in [41] a first global study about different types of android malware. They showed that normal applications were enriched with malicious content and found different apps containing similar malware code. Depending of this payload they grouped them in different families.

Besides the internal threat detection framework of AV companies, there exist also other open-source approaches that crawl various app-stores for detecting malicious applications. Lindorfer et al. [21] propose a framework for discovering multiple instances of a malicious Android application in a set of alternative application markets. Based on some lightweight indicators, such as the package name or the hash of an application, they found various malicious applications in different markets. DroidSearch [28] is another framework that crawls different app stores and stores for each application meta-data into a database. The database can be queried afterwards for detecting vulnerabilities or malicious applications.

Isolated environments for analyzing and detecting android malware are a well-established technique in the context of mobile malware analysis. Andrubis [20] or the Mobile Sandbox [32] are two examples. Usually, they use lightweigth static analysis techniques to find concrete malware patterns [5] in combination with a lightweight dynamic code analysis approach that monitors the application in a secure environment. The results are used to detect suspicious behavior or evaluate the risk factor [24] of an application. Due to the nature of the lightweight analysis, the proposed techniques reaches its limitations when it comes to sophisticated malware that triggers malicious behavior only under specific circumstances.

Signature based approaches [40] are a well-known techniques used by many anti-virus applications. Zheng et al. [40] proposed a new signature methodology that was able to easily discover repackaged malicious applications or even zero-day malware samples. Apposcopy, a tool proposed by Feng et al. [12] improves signature based approaches by a semantic based approach that specifies signatures that describe semantic characteristics of malware families. Both approaches rely on static information extracted from the bytecode. Hardening or even packers complicates the detection of malicious applications as shown by different researchers [29].

8 Conclusion

In this paper, we have described an investigation of a new malware family that infected more than 20,000 mobile devices in Korea. We described in detail the components of current state-of-the-art mobile banking trojans. Furthermore, we compared each individual technique of the malware with current state-of-the-art malware analysis techniques. Our results show that current malware poses many challenges to malware analysis techniques in order to trigger malicious behavior, showing the need for further research in this area. We furthermore demonstrated a new tapjacking attack that is exploited by the Android/BadAccents malware. It causes a security threat, as the user can be tricked into clicking/tapping on objects that trigger unintended behavior. The Android Security Team confirmed the attack and our proposed patch will be integrated in the next major release of Android.

References

1. Soot wiki, August 2014. `https://github.com/Sable/soot/wiki`.
2. Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.
3. Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In Axel Legay and Saddek Bensalem, editors, *RV*, volume 8174 of *Lecture Notes in Computer Science*. Springer, 2013.
4. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, June 2014.
5. Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit A. Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (Malware 2010) (MALWARE'2010)*, Nancy, France.
6. Carlos Castillo. Phishing attack replaces android banking apps with malware, June 2013. Blog.

7. Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
8. Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, New York, NY, USA, 2013.
9. International Data Corporation. Worldwide quarterly mobile phone tracker 3q12, November 2012. Blog.
10. Sebastin Guerrero D. Tapjacking-framework-for-android, 2012. Github.
11. 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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.
12. Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA, 2014. ACM.
13. Grossman J. Hansen R. Clickjacking attack, 2008. Blog.
14. Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, New York, NY, USA, 2013.
15. Simon Huang. The south korean fake banking app scam. Technical report, Trend Micro, February 2015.
16. Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, Los Alamitos, CA, USA, 1994.
17. Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, 2014.
18. Bitdefender LABS. Reveton / icepol ransomware moves to android. blog.
19. McAfee Labs. Threats predictions, 2015.
20. Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.
21. Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. AndRadar: Fast discovery of android applications in alternative markets. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, volume 8550, London, UK, July 2014.
22. Denis Maslennikov. Zeus-in-the-mobile - facts and theories, October 2011. Blog.
23. Marcus Niemietz and Jörg Schwenk. Ui redressing attacks on android devices, 2014. BlackHat Asia.
24. NViso. `http://apkscan.nviso.be/`.
25. Board of Governors of the Federal Reserve System. Consumers and mobile financial services 2014, March 2014.

26. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, New York, NY, USA, 2014.
27. Siegfried Rasthofer. Codeinspect says "hello world": A new reverse-engineering tool for android and java bytecode, December 2014. Blog.
28. Siegfried Rasthofer, Steven Arzt, Stephan Huber, Max Kohlhagen, Brian Pfretschner, Eric Bodden, and Philipp Richter. Droidsearch: A tool for scaling android app triage to real-world app stores. In *Proceedings of the IEEE Technically Co-Sponsored Science and Information Conference 2015 (SAI)*, July 2015.
29. Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, New York, NY, USA, 2013.
30. Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4), December 2011.
31. Raimondas Sasnauskas and John Regehr. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, New York, NY, USA, 2014. ACM.
32. Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, New York, NY, USA, 2013. ACM.
33. statista, jul 2014. `http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`.
34. Symantec. Symantec report on the underground economy, 2008.
35. Android Developer Team. Device administration. `http://developer.android.com/guide/topics/admin/device-admin.html`.
36. Android Developer Team. Device policymanager. `http://developer.android.com/reference/android/app/admin/DevicePolicyManager.html`.
37. Android Developer Team. Intent. `http://developer.android.com/reference/android/content/Intent.html`.
38. Emre Tinaztepe, Doğan Kurt, and Alp Güleç. Android obad. Technical report, COMODO, July 2013.
39. Yuanyuan Zeng, Kang G. Shin, and Xin Hu. Design of sms commanded-and-controlled and p2p-structured mobile botnets. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, New York, NY, USA, 2012. ACM.
40. Min Zheng, Mingshen Sun, and John C. S. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '13, Washington, DC, USA, 2013. IEEE Computer Society.
41. Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.