(In-) Security of Smartphone AntiVirus and Security Apps

Stephan Huber, Siegfried Rasthofer, Steven Arzt
Fraunhofer SIT
Darmstadt, Germany
{stephan.huber, siegfried.rasthofer, steven.arzt}@sit.fraunhofer.de

ABSTRACT

Android is the by far most popular operating system for smartphones today. Many people entrust their Android-based phone with highly sensitive data such as business documents and credit card information, or perform critical tasks such as online banking on their devices. To protect their devices against threats from malware or attackers who aim to exploit security vulnerabilities, many users rely on AntiVirus and security apps available from renowned vendors.

In this paper, we show that those apps, however contain severe vulnerabilities on their own and installing them can even decrease the overall security of the device. We analyzed the most frequently-downloaded security apps and found that they were vulnerable for remote code execution and malware database downgrades. Some AntiVirus scanners could be disabled remotely without the user noticing, or devices could be locked and wiped remotely without proper authentication. We show that security apps are of no better quality than regular apps when it comes to their own code security.

1. INTRODUCTION

Android is one of the most popular smartphone operating systems today. In 2015, it had a market share of more than 80% according to the Worldwide Quarterly Mobile Phone Tracker[1]. This popularity is, to a large extent, driven by a large ecosystem that offers a wealth of apps for every need. The official Google Play Store alone offered more than 1.6 million apps in 2015. Not all of these apps are benign, though. Some apps are created with malicious intent to steal the user's sensitive information or access premium services at his cost. Since many users perform sensitive tasks such as online banking on their phone and many devices are equipped with privacy-invasive sensors such as GPS, this constitutes a high risk. Additionally, some apps are not outright malicious, but suffer from severe security vulnerabilities. These vulnerabilities allow malicious apps installed on the same device or even remote attackers to steal sensitive data or manipulate the device.

To protect their device and data from attacks, be it through malicious apps or through exploits for security vulnerabilities, many users rely on security apps from renowned vendors. Just like their desktop counterparts, these apps offer services beyond traditional anti-virus scanning. Besides checking the list of installed apps for known unwanted software, these apps block the browser from displaying phishing or otherwise malicious websites, and offer to block or wipe the device in case it gets lost or stolen. While all these features are useful and intended to enhance the security of the device, they, again, are developed by human app developers who can make mistakes. Previous studies have found [22, 7, 8] that many Android developers have either not had proper security training or, at least, do not employ rigorous security testing and auditing on their apps. In this paper, we therefore investigated whether these security apps published by major vendors share the same programming mistakes and security vulnerabilities that are prevalent in many other Android apps.

Our results show an alarming number of severe vulnerabilities in security apps that can allow attacks such as remote code execution, virus signature downgrades, disabling the security app without the user knowing, or remotely locking or wiping the user's device without requiring any credentials or secrets. In summary, installing some of the tested apps *decreased* the overall security level of the device instead of further protecting the user. For all of our findings, we went through a responsible disclosure process in which we gave the respective vendors 90 days of time to fix the issue. All but one company worked with us to secure their products, even changing their own infrastructure to improve security. In summary, this paper presents the following original contributions and findings:

- A security analysis of popular security apps for Android, including the ones from Avira, McAfee, ESET, Kaspersky, Malwarebytes, Androhelm, Cheetamobile.
- A set of popular attack vectors in security apps that developers need to protect against, and
- The conclusion that developers of security apps are not more security-aware than any other app developers.

The remainder of this paper is structured as follows: In Section 2, we give some background information on common features of security apps for Android. In Section 3, we describe our attacker model and explain the possible interactions between the adversary and the security app, before showing the actual vulnerabilities and attacks in Section 4. Our responsible disclosure process is laid out in Section 5. Afterwards, we discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

[1]http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37

Current AntiVirus or security applications for Android do not only protect the user against known malicious applications with their malware scan engine, they also come with additional security features, which protect the user against malicious or unwanted activities. Each security application contains different security features. In the following, we will explain the most common ones. All of these features can, however, only work if they themselves cannot be manipulated, deactivated, or even exploited for malicious use. We therefore derive security requirements for these features to ensure proper functionality.

2.1 Malware Detection Engine

Every AntiVirus application gets shipped with a virus scan engine, which decides based on a signature database whether an application is malicious or not. The app frequently downloads updates for this database from the respective vendor's update server. This update mechanism works without any user interaction and automatically downloads not only signature files, but also scan engine updates in the form of executable code files or library code .

Requirements: The scan engine itself must fulfill the concept of availability for a permanent protection. The files downloaded by the update engine need to fulfill the concepts of integrity and authenticity to make sure that no unauthorized or tampered virus databases or code files can be installed. Most of the AntiVirus signature files are also encrypted to ensure confidentiality due for intellectual property concerns. Furthermore, availability of the update server must be guaranteed to ensure that users are always able to update their scanners.

2.2 Spam Protection

The spam protection feature blocks unwanted SMS spam messages or spam phone calls.

Requirements: The blocked strings of SMS messages or phone numbers shall fulfill the concepts of availability and integrity for protection against manipulation. There should not be a way to bypass the black or whitelisting of the protection system. Furthermore, these lists must be kept confidential, as especially whitelists can contain sensitive user data, e.g., telephone numbers of interest to the user.

2.3 Secure Browsing

The secure browsing module is monitoring the browser in order to prevent access to malicious websites the user might otherwise inadvertently visit. The filter is usually implemented using a local proxy that is running in the context of the AntiVirus app. The web browser is then configured to route all of its traffic through that proxy. For each access, the app checks the requested URL against a backend service run by the AV vendor. An alternative technique that does not require a local proxy continuously polls the browser history. When the user navigates to a website in his browser, the respective URL will appear at the top of his browsing history. The security app reads it from there and sends it to the AV vendor's backend. Regardless of how the browser is observed, the AV app must display a warning message and prevent the web page from being loaded if the respective URL has been found on the blacklist and is considered malicious. For this blocking step, the AV apps usually contain special web pages either inside the app or hosted on the AV vendor's backend to which the browser is redirected instead of visiting the original, malicious URL.

Requirements: The confidentiality and integrity of the URL entered by the user has to be protected while it is submitted to the Antivirus backend. The secure browsing module needs to fulfill the concept of availability so that all requested URLs can indeed be checked. Furthermore, the protection must not interfere with the user's surfing behavior on benign websites. The blocking function of the secure browsing module must also be reliable, a malicious website should not be able to circumvent it.

2.4 Device Configuration Advisor

The Android OS offers different security settings, such as "allow app installation from unknown sources", "screen lock type (pin, password, pattern, etc.)", "device encryption", etc. For maximizing device security, these settings must be configured properly by the user. Therefore, many security apps contain a module which scans the phone for insecure settings and gives recommendations for improving the device configuration.

Requirements: The authenticity and integrity of this module or its database of configuration rules must be guaranteed so that an attacker cannot inject weak configurations that are then proposed to the user.

2.5 Privacy Advisor and Protector

The privacy advisor module protects the user against privacy related issues such as data leaks. If the user's phone gets lost or stolen, he can, for instance, send a special SMS containing a secret token or password to the phone. Upon receipt of such a special SMS message, the security app locks or wipes the devices to prevent data exfiltration or to simply make the device unusable for the thief. Other apps can be queried remotely using similar SMS commands to retrieve the GPS location of the device in an attempt to ease recovering the device or identifying the thief.

Requirements: The privacy protector module actively triggers protection mechanisms that can have side-effects such as data loss or phone locking. Therefore, the mechanism must be authenticated so that it can only be triggered by authorized users. Furthermore, once triggered, an attacker shall not be able to bypass the device lock or recover the data from the wiped device.

2.6 Premium Upgrade

Many AntiVirus vendors offer a free basic version of their tools. If the user needs more functions, he can pay and unlock premium features. Common premium features include additional protection modules or more frequent updates to the signature database.

Requirement: An attacker shall not be able to upgrade the application without any payment. Note that this requirement is different to the previous ones, because it protects the vendor rather than the user.

3. ATTACKER MODEL

Our attacks presented in Section 4 are constructed with respect to an attacker model. We assume that the security apps are installed from the official Google Play Store, i.e., the app itself is untampered and was up-to-date at the time of our study. We also assume the phone not to be rooted, i.e., Android's default protection mechanisms such as app isolation are in place, are fully functional, and cannot be

circumvented. For our analysis and the proof of concept attacks we used Google Nexus 4 and Nexus 5 devices. We, however, assume our attacks to be device-independent.

In addition to these basic definitions that hold for all of our attackers, we distinguish different attacker models. Each definition shows the power of the attacker and its limitations. Each of the defined models can operate as an *active* or *passive* attacker. The *passive* attacker can read all data transmitted over external channels such as Wi-Fi or GSM. The *active* attacker has the capabilities of the *passive* attacker, but can also manipulate data in transit.

- A **remote attacker** can only remotely interact with the device, but has no physical access to it. The communication channel does not matter, it can be via email, web or sms. The attacker can, for instance, manipulate a website with malicious content or send a prepared link to the smartphone user. When the user inadvertently opens the website or clicks on the link, the attack is triggered. Note that user "cooperation" is not necessarily required, e.g., in the case in which the attacker only needs to send a specially-crafted SMS message that is directly processed by a vulnerable app.
- A **local attacker** has physical access to the device. She can install or remove applications from the device or change configuration settings. Note that this attacker still does not have root access and is bounded by Android's default security mechanisms such as app isolation.
- The **man-in-the-middle attacker** (mitm) is a special form of remote attacker. The *passive* mitm adversary is able to eavesdrop information without the victim's knowledge. She cannot modify the traffic. The *active* attacker is able to eavesdrop, redirect or manipulate the network traffic without user knowledge. In case of a secure SSL connection she is not able to break a secure scheme, but she makes usage of any kind of well-known vulnerability [6, 1, 2, 19].
- An **attacker app** is downloaded and installed by the user. Most likely, the user was tricked into installing the application in some way. The app does not need root privileges and runs in the same context as other benign applications. It is designed to abuse a specific known vulnerability in the target app.

4. ATTACK SCENARIOS

Based on the different attacker models described in the previous section, we now explain various vulnerabilities in the tested security apps and show proof-of-concept attacks that exploit them.

4.1 Derived Attack Targets

In our analysis we consider Android antivirus and security applications from the Google Play Store. All of the apps we considered in this study are listed in Table 1 together with the respective versions we tested. From the security requirements defined in Section 2 combined with our specified attacker models (see Section 3), we derived the following attack goals:

- Unlock non-free premium app features without paying.
- Steal license keys or credentials from other users.
- Disable the AntiVirus scan function and real-time monitoring.
- Lock or wipe the device remotely without authorization.
- Remotely inject and execute code so that it runs in the context of the security app.

The following subsections describes successful attacks on the security apps that achieve all of the aforementioned goals. For each attack, we describe the concrete vulnerability that made the attack possible. Where applicable, we show excerpts of the vulnerable code.

4.2 Financial Fraud

The first class of attacks is targeting the business model of the app vendors, but does not adversely affect the end-user of the app. Each analyzed application provides premium features like faster update, a secure browsing function, or other special features that extend the normal functionality of the respective app. After a certain test period, the user is either limited to the basic features or has to pay for continued access to these premium features. If a user is able to bypass the payment check, he can use the premium features for free.

We analyzed the implementation of the apps' license validation systems in order to verify if it is possible to permanently unlock the premium features without any payment. The first implementation flaw in one of the examined security applications was a client side verification of the license. The verification is just handled by an entry in the `SharedPreference` file named `boolean name="isPro"`. The app checks if the value associated with this key is set to `true`. If this is the case, all premium features of the application can be used. There is no server side cross verification if the user has any valid license.

Normally, the shared preferences of an app are stored in a file located in the app's private data directory. Due to Android's app isolation, only the user account under which the respective app is executed has the permission to access the app's private data directory and thus the shared preferences file. However, an adversary (most likely the legitimate app user in this case) can manipulate this shared preference entry, without requiring root privileges, by using Android's backup mechanism. By activating the Android USB debug function on the device [10], it is possible to backup the application using the `adb backup` function. Afterwards the backup file contains the local app folder data, which can be manipulated on the PC and afterwards restored to the device [21]. This means an attacker can change the `SharedPreference` value from `false` to `true` in the backup archive and restore the changed file. This is possible, because the app did not disallow backups in its manifest file. The described attack was possible on the Androhelm AntiVirus application.

Another possible scenario causing financial fraud is an attack that steals the AntiVirus license key or user credentials from another user. Such an attack was possible in the ESET Mobile Security & Antivirus application by a man-in-the-middle attack. The application transfers the user credentials and license key to the vendor's server for verification. For this connection, the app uses SSL, but due to a faulty implementation, the connection can easily be intercepted. The

Name	Version	ID
AntiVirus app (AndroHelm)	1.6	com.androhelm.antivirus.free2
Avira Antivirus Security for Android	4.2	com.avira.android
CM Security	2.7.3	com.cleanmaster.security
Mobile Security & Antivirus	3.2.4.0	com.eset.ems2.gp
Kaspersky Internet Security for Android	11.9.4.1294	com.kms.free
Malwarebytes Anti-Malware	2.00.3.9000	org.malwarebytes.antimalware
McAfee Security & Power Booster	4.5.0.601	com.wsandroid.suite

Table 1: All security applications considered in this study. The source of each app is `https://play.google.com/store/apps/details?id=ID` with ID replaced by column entry

application overwrites the default `TrustManager` of Android in a wrong way, as described in [6], and does not verify the server certificate. This allows a man-in-the-middle attacker (see Section 3) to sniff the connection by impersonating the server with a self-signed certificate. An excerpt of the app's registration process in which credentials are transmitted is shown in listing 1.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 ...
3 <NODE NAME="LicenseUsername" VALUE="Fdax6a7wj/I+ZEet"
      TYPE="STRING"/>
4 <NODE NAME="LicensePassword" VALUE="Fdax6a7wj/I="
      TYPE="STRING"/>
5 <NODE NAME="PreviousLicenseUsername"
      VALUE="JNaV6Yvw1vJrZAStgsgqddgxE7x6+OkMc9013Q=="
      TYPE="STRING"/>
6 <NODE NAME="PreviousLicensePassword"
      VALUE="VNa26a/wzvIjZAetwsgtdY4xGLw="
      TYPE="STRING"/>
7 ...
```

Listing 1: Excerpt of ESET login request

Besides the faulty SSL protection, the application uses an additional base64 encoded encryption scheme for the license data (line 3-6 in listing 1). After analyzing the crypto algorithm, we determined that the credentials are just XOR-ed with a pseudorandom generated stream key. To get the key values, an attacker can use a chosen-plaintext attack. The knowledge of the XOR encryption key value combined with the faulty SSL implementation allows a mitm attacker to steal and decrypt the license information. This allows an attacker without an own license to use a victim's license instead to avoid own payments.

4.3 Denial of Service Attacks

As defined in Section 2, a security app and especially an AntiVirus tool must constantly be available, i.e., run in the background and detect potential malicious behavior. Whenever new apps are installed or downloaded, the scanner must compare them to the malware database and warn the user in case of a finding. Due to improper exception handling, an attacker is, however, able to crash some of the security apps, and consequently, render the protective features unavailable.

In many cases, these vulnerabilities arose due to improper `Intent` handling. Sending an improperly-formed intent to the app triggered an exception that remained unhandled and thus terminated the app or at least important parts of it. An Android `Intent` can be described as a messaging object[13, 12] containing information for inter-process communication. The different Android components of an application, for instance a `BroadcastReceiver` or an `Activity` can receive such an object for passing information from the system or other applications to the component. After receiving

```
1 public void onReceive(Context c, Intent intent) {
2  Bundle bundle = intent.getExtras();
3  if(bundle != null) {
4   Object o = bundle.get("pdus");
5   ...
```

Listing 2: Faulty onReceive() method

```
1 public void onReceive(Context c, Intent intent) {
2  if (intent != null) {
3   Bundle b = intent.getExtras();
4   if (b != null) {
5    Object[] pdus = (Object[]) b.get("pdus");
6    SmsMessage msg =
       SmsMessage.createFromPdu((byte[]) pdus[0]);
7    String phoneNumber =
       msg.getOriginatingAddress();
8    //if phoneNumber contains a non-integer value,
       an exception will be thrown
9    int number = Integer.parseInt(phoneNumber);
10 }
```

Listing 3: Intent value cast exception

such an object, the application can extract the encapsulated information from the `Intent`. The most common flaws we observed in the security apps were missing checks on data availability (i.e., does the intent contain a specific expected data item) and type conversion errors. Most of the time, one could simple crash the application with an Intent object that was `null`. In listing 2, we show a vulnerable implementation of a `BroadcastReceiver` component. The code example acts upon receiving an `Intent` object as a parameter of the `onReceive(Contextc, Intent intent)` method.

If the implemented method receives a `null`-Intent for the `intent` the `getExtras()` method is called on `null`. This causes a `NullPointerException`. Due to a missing `Exception` handling, this exception propagates up to the Android operating system and crashes the application. Since Android version 4.4 (KiKat) the OS will filter `null`-Intents for mitigating such flaws. In Listing 3, we show a second example, this time based on improper type conversions. In this case, the developer assumes that the incoming value, which is supposed to be a telephone number, consists only of numbers (see line 9). Consequently, he performs a type cast on the number without any further validation. If the content contains a non-integer value, an `Exception` is thrown and the application crashes because of missing Exception handling.

For exploiting these vulnerabilities, an *attacker app* installed on the user's phone can trigger such exceptions by sending malformed Intents to the security application. Since all of our analyzed security applications listened to ex-

ternal Intents, i.e., Intents sent from another application, the denial-of-service attack was possible *on all of these apps*. We found such vulnerabilities in the following security applications:

- Androhelm AntiVirus App
- Avira Antivirus
- Malwarebytes Anti-Malware
- McAfee Security

4.4 Remote Attacks

Remote attacks are one of the most severe types of attack on a smartphone. We defined two types of remote attackers in section 3. A generic remote attacker can send SMS messages, e-mails, etc. to the device, but cannot interfere with or read any normal traffic. A more specialized form, the mitm attacker, on the other hand, can also interfere with existing traffic. Referring to this model we describe three different possible types of remote attacks found in the analyzed smartphone security apps.

4.4.1 SMS Based Attack

The first class of attacks is based on malformed SMS messages. Recall that some apps allow the user to remotely wipe or lock a device in case it gets lost or stolen. To trigger this function, the user needs to send a specially-formatted SMS message to his phone. For security reasons the app user can specify an authentication password and a friend's phone number from which she can send the command message. The security app will discard messages with invalid passwords as well as messages from other phone numbers than the explicitly allowed one. The latter protection is, however, ineffective. SMS sender numbers can easily be spoofed. Furthermore, the required password is stored on the phone in plain text. If an attacker is able to read out the respective file through a malicious app, for instance, he can afterwards remotely wipe the device at any time, even if his app used for exfiltrating the password lacks the required permissions. As a result, the attacker can abuse the malware scanner application as a ransomware [30] by remotely locking the device unless the user pays a specific amount of money.

While this attack scenario still requires the attacker to first exfiltrate the authentication password, we found one security app that contained a vulnerability that allowed us to bypass the password check without any password exfiltration. This is usually due to improper input verification on the incoming SMS message. In one app, the SMS-based protocol for executing commands such as "wipe" and "lock" was defined as `SMS_PASSWORD [SPACE]command` with `command := wipe | lock | locate | callme | unlock | siren`. The commands trigger several functions on the device, for instance wipe the device, lock the device or send location information back to the owner. We analyzed the code that is responsible for handling received command SMS messages in detail. The message is parsed and tokenized by the `split`-function[2]. More precisely: It is splitted on each [SPACE] sign (`split(" ")`) of the message to obtain a list of tokens.

The password verification logic is described in pseudo code form in listing 4. The `PASSWORD` value is a user defined password stored in a configuration file of the app. If the user triggers a remote SMS command, the `SMS_PASSWORD` token is stored in `sms_cmd[0]` field and compared to this `PASSWORD` value (first part of line 3). If it matches, the corresponding `command` is executed. In the default configuration of the app, there is, however, no password defined. This means the password is an empty string by default. This is not checked. Therefore, an attacker can trigger commands by sending an SMS message with an empty password unless the user has actively changed the app's configuration and has set a non-empty password. Listing 4, shows the issue in line 3.

```
1 ...
2 sms_cmd:=sms_msg.split(" ")
3 if sms_cmd[0] == PASSWORD OR sms_cmd[0] == ""
4    execute(sms_cmd[1])
5 ...
```

Listing 4: Pseudo code of the SMS command handling logic

If a remote attacker achieves to send an SMS like `[SPACE] command[SPACE]somestring` it will be tokenized by the code and the password part will be interpreted as empty. Compared with the empty password the condition will be true and the application will accept the received command and trigger the function. This can be realized by hand-crafting a SMS message and replace the phone number by a space character. The number field of a SMS message can be modified and is not required to be in any specific format for the SMS message to be transmitted across the GSM network and to finally reach the victims device [5, 32]. An attacker who knows the victim's phone number, can remotely lock or wipe the victim's device. Furthermore, this attack can also work locally without sending an actual SMS message over the network.
A malicious app installed on the same device can simulate such a malformed SMS message. The malicious application doesn't even need any suspicious API calls that could hint at its malicious behavior, making it easier for the attacker to distribute his malware to unsuspecting users. The malicious app could then implement a simple ransomware scheme: Unless the user pays a specific amount of money, his phone gets locked with the help of the security app, which acts as a confused deputy. Because of these security issues, most other security apps no longer use SMS-based control techniques, but instead forward such commands via Google Cloud Messaging [11].

4.4.2 Web Based Attacks

Another remote attack was found in the McAfee Security application. The attack is web-based and works like a drive-by attack. This means the attacker does not have to compromise the victim's device directly. It is enough to add some malicious code into a web page and once the victim accesses the website with her smartphone, the code will be executed. This works because of a vulnerability in the secure browsing module provided by the security application. Normally, the task of this module is to prevent the user from inadvertently accessing malicious websites. In this case, the module even allows the attacker to inject malicious code in its own context.

The secure browsing module knows the current URL a user wants to access. This is, for instance, technically pos-

[2]https://developer.android.com/reference/java/lang/String.html#split\%28java.lang.String\%29

```
1  <script type="text/javascript">
2  //URL with attacked JavaScript injection
3  var lksu = "http://evil.com/";alert("XSS");var
       x="";
4  if (lksu != "") {
5    document.getElementById('gobackbtn').href =
       lksu;
6  }
7    else {
8    if (history.length <= 1) {
9      divelem = document.getElementById("gobackbtn")
10   if (divelem) {
11     divelem.style.display = "none"
12   }
13 }
14 ...
```

Listing 5: Code excerpt of warning page

sible by accessing the browser history or by implementing a local proxy server. This URL is encrypted and transferred to the AV vendor's backend server where it is compared to database of known malicious websites. If the URL is detected as malicious, the secure browsing module will block the access to the website and shows a warning page that informs the user of the issue. In this particular app, this warning page was vulnerable to a cross site scripting attack. The blocking page shows the full URL of the suspicious site. It also displays a link with which the user can continue to the blocked site nevertheless in case he chooses to accept the risk. The app's warning page, however, also used the URL of the malicious website inside JavaScript without further validation. The code of the warning page and the embedded URL with JavaScript attached is shown in listing 5 line 3.

An attacker or a malicious site which is aware of this flaw can design a page whose malicious behavior cannot be blocked. If his site is not on the blacklist, his malicious code is directly triggered when a user accesses his page. If the site is blacklisted, he can still execute his malicious code through the vulnerability in the blocking system itself, even though the user cannot visit the original malicious site. Figure 1 shows the blocking page with a proof-of-concept exploit. We attached JavaScript code to a URL pointing to a suspicious phishing web site URL taken from phishtank[3].

In our study we did not evaluate the full power and consequences of an XXS attack on the Android browser. This was already done in previous research [4, 26]. Some hypothetical scenarios could be: phishing sites aware of this vulnerability can for instance disable or bypass the blocking function of the secure browsing by calling the browser's back function via JavaScript. Recall that some security apps do not provide a local proxy, but instead monitor the user's browsing history. These apps can only navigate the browser to the warning page after the blocked malicious page has already been added to the history. Triggering the backwards navigation will thus make the browser return to the malicious page that should originally be blocked, effectively circumventing the secure browsing module. Additionally the injected code can load further external JavaScript code that tricks the user into installing a malicious application or that redirects him to a malicious website which is not detected by the secure browsing.

4.4.3 *Man-in-the-middle Attacks*

[3]http://www.phishtank.com/

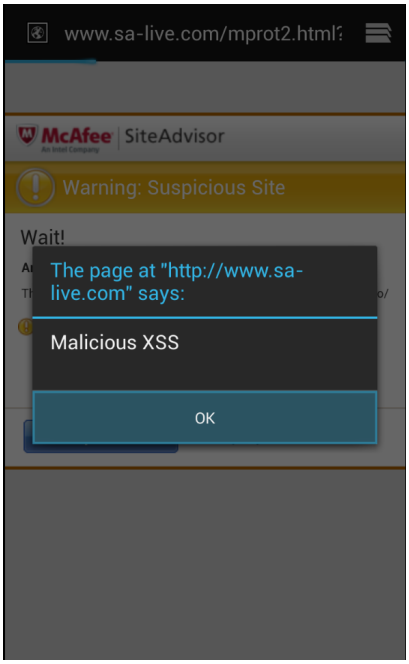

Figure 1: XSS Attack on secure browsing feature

We also found serious remote attacks, which require a mitm attacker. Many security apps use unauthenticated and unencrypted plain `http` connections to communicate with the backend of the respective AV vendor. Most of the analyzed applications also used this plain channel for transferring sensitive data such as virus signatures or scan engine updates. Therefore, confidentiality, integrity and authenticity which would have been provided by mechanisms such as SSL were not guaranteed by the communication channel. This allowed a mitm attacker to redirect the traffic to his own server instead. By exploiting this vulnerability, we were able to downgrade the signature database of multiple AntiVirus applications.

In an attempt to mitigate the risks, the developers implemented their own integrity and confidentiality mechanisms that did not follow commonly accepted practices. In general, implementing own crypto schemes does not follow the well-known secure coding guidelines and is certainly not advisable. We therefore especially looked at those security apps that implemented their own ciphers for encryption.

The update process of the Malwarebytes application, for instance, downloaded its malware signature files via plain `http`. The downloaded files were encrypted using the symmetric RC4 encryption scheme. This approach mistakenly assumes that encryption implies authentication and integrity protection. Even if we assume that an attacker must be able to decrypt the file in order to manipulate the virus signatures contained in it, the approach is still faulty. RC4 is a weak scheme against which powerful attacks are known to literature [17, 16, 9]. Worse, the downloaded signature updates could even be decrypted without breaking the scheme, because the developers had hard-coded the *encryption key* directly into the application's code. An attacker could simply reverse-engineer the application, extract the key, and decrypt the files. In summary, the attacker redirect the plain `http` traffic to his own server where he offers tampered virus signature databases that are either old or from which cer-

tain signatures of interest have been removed. The app will not be able to detect the tampering due to the misconceptions and vulnerabilities explained above. As a consequence, an unsuspecting user might even decrease the security of his device by updating his security app if he falls victim of the described attack. After we have reported these issues to the developers of the vulnerable app, they informed us that this problem had already been reported earlier for the Windows version of their AntiVirus software[4]. This example shows that such vulnerabilities not only affect the apps, but sometimes also implementations of the same app for other operating systems due to code re-use.

Other apps such as CM Security and Avira AntiVirus Security implemented an integrity check for detecting manipulations to the downloaded update files. In the CM Security app, the integrity verification was based on MD5 hashes. The app first downloaded a central `ini` file containing the expected MD5 sum for every file to be updated. An attacker who wants to manipulate or replace the update files thus simply needs to first replace the MD5 sum in this `ini` file with an updated value. This is easily possible, because the `ini` file itself is unprotected. Furthermore, even if it were protected, MD5 is no longer considered to be a safe hash algorithm.

One of the files updated by the app was an executable file for the scan engine. After the update completes, the app automatically executes the new version of the executable. An attacker who is able to replace or manipulate this executable therefore automatically gains the possibility to execute arbitrary code in the context of the security app. Therefore, the injected code inherits all permissions of the security app which can be quite substantial. Recall that security apps often need a large amount of privileges because of their additional features (wipe/lock, safe browsing, etc.)

The Avira AntiVirus Security application uses a slightly different integrity verification system. At the beginning of the update process, the application loads an unprotected root init configuration file (see listing 6) containing a list of all files required for the update. Such files are further configuration files, malware signature database and scan engine libraries.

```
1  ...
2  <FILE>
3    <NAME value="axvdf/common/int/libaescn.so"/>
4    <FILEMD5
       value="93dbbc5000427a4f513a05480b5ff1a6"/>
5    <PEFILEMD5
       value="93dbbc5000427a4f513a05480b5ff1a6"/>
6    <FILESIZE value="1228"/>
7    <ZIPMD5
       value="e707a871b70eba16482be4374e94a329"/>
8    <ZIPSIZE value="477"/>
9    <OS value="ALL"/>
10   <VERSION value="8.1.8.4"/>
11 </FILE>
12 ...
```

Listing 6: Excerpt from androideabi-int.info file

The `.info` file also contains the corresponding file size, version number and hash values for the files (see listing 6). Each file listed in the `.info` file will be downloaded in the given order and stored in a temporary folder in the app directory. All listed files, except for the `.info` file itself, have an additional integrity verification. If the verification of all files succeeds, the files are copied to their proper places in the app directory, replacing the respective old versions of the file. If, on the other hand, a single file fails the integrity verification, the update process is aborted altogether, all downloaded update files are deleted from the temporary folder, and the old versions of the target files are kept.

The integrity check of each file based on asymmetric cryptography. This means the files are signed using a private key only known to the vendor of the security app. The corresponding public key is hard-coded into the app for signature verification. This allows the app to ensure integrity and authenticity for all downloaded files. However, this protection only applies to the individual files and not to the communication channel itself. Because of the missing server authentication (`http`-connection), an attacker is able to redirect the traffic to a fake update host which serves manipulated files. As long as these files are properly signed, the app cannot detect the attack. An attacker can thus, for instance, perform a downgrade attack and serve outdated virus signatures through his fake web server. The scanner would then no longer be able to detect current malware, though the user is informed that the virus signatures have been updated properly.

For further attacks, we did not find a way to bypass the signature verification checks. Therefore, we could not manipulate actual content of the data files aside from downgrades. Adding new data files was not possible either, because these files would have had to be signed as well. However, recall that the `.info` file is not signed and can thus be manipulated on the fake host. Furthermore, all files are signed with the same key. This allows an attacker to take an existing, signed file and have it take the place of a different file with a different semantics, effectively rending the target file unusable. The `.info` file contains two conceptual sections: One for data files (virus signatures, etc.) and one for executable files such as the scan engine libraries. Our next attack thus replaces one of the library files with a data file from the other section. Since both files are signed with the same key, the signature verification still succeeds and the file is overwritten by the update engine. However, when the scan engine is started, it cannot run the new "library" file anymore, because it's not a valid executable. This leads to an error message being written to the console. Due to a second flaw, this error was, however, not forwarded to the user interface of the app, leading to a worst-case scenario: The scanning progress bar is still running and showing a valid scan process although the underlying scan engine is not working any more. The user thinks actual progress is being made in the scanner, but in fact, nothing happens and the scan will always report a clean device, even in the presence of malware.

Another mitm attack was found in the Kaspersky Internet Security application. We were able to remotely execute arbitrary code by manipulating a zip file together with a path traversal injection [24]. Just as in the cases described above, this app is also using an unauthenticated `http` communication for downloading updates for the scan engine and the virus definition files. All files are downloaded into a temporary folder inside the local app directory. Again, there is an integrity check on the individual files. Only if this check succeeds, the respective file is moved into its respective target folder. There are, however, two `zip` archives that are not integrity-protected. These files contain advertisement data

[4]https://bugs.chromium.org/p/project-zero/issues/detail?id=714

as web pages (.html, .jpg, .css and a .xml file). As the download happens through an unprotected `http` channel, a mitm attacker is able to manipulate these `zip` files, e.g., inject additional code or files. During the app's update process, advertisement `zip` files are extracted into a predetermined folder for advertisement content. The attacker then faces the challenge that can add executables such as `dex` or `jar`, but needs a second vulnerability to actually execute them.

The easiest approach is to replace an existing code file that is normally loaded by the security app. The Kaspersky app uses reflection to load code from a file called `pdm.jar` file, which is regularly updated through the update channel. Recall that this file is a normal code file that is integrity-protected during the update process. Therefore, it cannot directly be replaced. However, since the code for unpacking the unprotected advertisement `zip` archives is vulnerable to a path traversal attack similar to Welton's work [25], the attacker can trick the app into overwriting the `pdm.jar` file while unpacking the advertisements. The attacker inserts an additional `pdm.jar` file into the manipulated `zip` archive and renames it to `../../../../../../../../../../../../../../../../../../../../../../../../data/data/com.kms.free/app_bases/pdm.jar`. Once the file has been overwritten, the attacker's code will be executed upon the next restart of the security app. Recall that security apps in general need to define a lot of sensitive Android permission due to their security features (see Section 2). An attacker who is able to execute arbitrary code in the context of such an app gets nearly full control of the smartphone. If the injected code also contains a root exploit she can reach full remote control over the victim's device.

5. RESPONSIBLE DISCLOSURE

We reported all of our findings to the respective app vendors through secure channels, usually PGP-encrypted e-mails when available. Some vendors did not list their public PGP keys on their webpage and we had to first contact them to obtain the keys before being able to then report the actual vulnerabilities. In some cases, PGP keys were available on the respective vendors' web pages, but were expired. In this case, we also had to first obtain a proper key. We gave all vendors 90 days to fix the vulnerabilities in their apps before publishing any of the findings. To the best of our knowledge, all vulnerabilities, except one, are fixed by now.

6. RELATED WORK

Kantola et al. [15] and Chin et al. [3] analyzed the security implications of inter-app communication in Android. They especially showed that missing or improper intent validation on the side of the receiver can lead to vulnerabilities exploitable through malformed intents sent by other apps installed on the same device. We found several instances of this issue in the security apps that can be used for disabling the scan engine.

Fahl et. al [6] examined how Android apps implement SSL communication. They developed a framework for discovering vulnerabilities arising from improper certificate validation in Android applications. We discovered several such flaws that allowed an attacker to impersonate the AntiVirus vendor's backend server against the app. Fahl et. al also manipulated the signature files of the Zoner Antivirus Free app during the update through a mitm attack. A vulnerability in the SSL verification allowed them to change the downloaded virus signatures so that the AntiVirus application detected itself as malware and recommended to delete itself. Min et. al also describe attack vectors [18] in the update routines of well-known desktop antivirus solutions. According to their findings, unprotected updated allow malware authors to manipulate the protection software and inject code into the antivirus application. As AntiVirus tools often run with elevated privileges, the injected code is executed with SYSTEM user rights, giving the attacker full control over the user's device. Korent [14] uses fuzzing to find implementation flaws in desktop antivirus solutions. Some of the implementation flaws could be exploited.

Rasthofer et al. [27] and Niemietz et al. [20] presented tapjacking/ ui redressing attacks on the Android system and on individual apps. Several apps in our analysis were vulnerable to this kind of attack as well.

Simon [29] et. al analyzed the anit-theft features like "remote wipe" or "remote lock" services on Android AntiVirus and security applications. They already found that such features were "unreliable due to poor implementation practices". We were able to confirm that such vulnerabilities exist in many security apps. Additionally, Laurent et al. [28] described that even after a wipe or a factory reset, most of the data can be recovered, essentially rendering these security features useless. These limitations, while initially discovered for local built-in wiping features, also apply to remote wiping capabilities of security apps.

On more general terms, Rastogi et. al [23] tested the accuracy and reliability of scan engines for detecting modified malware. The resulting framework DroidChameleon modifies Android malware with different obfuscation techniques and evaluates if anti-malware apps still detect these apps as malicious. The Google Project Zero[5] member Tavis Ormandy discloses vulnerabilities [31] in several desktop antivirus and security products. One of the findings in the Malwarebytes application for Windows, was equivalent to the vulnerability we independently discovered in the Android version.

Antivirus application test institutes like AVTEST[6] or AV comparatives[7] evaluate the detection rate of the AntiVirus scanners and list additional security features of antimalware and security applications. They, however, do not evaluate the implementation of the security application itself. Our study was the first one which evaluated the security of the Android AntiVirus and Security apps in details and approved the found vulnerabilities with proof of concept exploits.

7. CONCLUSION

In this work, we presented a comprehensive security analysis of common Android AntiVirus and security applications. Current security applications do not only offer a malware scan engine, they also provide different security features like secure browsing or anti-theft. However, we found serious vulnerabilities in all of the tested applications. In a simple case, we would have been able to harm the app vendor's business model by upgrading a trial version into a premium one at no charge. In other cases, attackers would be able to harm the end-user by completely disabling the malware-scanning engine remotely. Broken self-developed crypto implementa-

[5]http://googleprojectzero.blogspot.de/
[6]https://www.av-test.org
[7]http://www.av-comparatives.org/

tions or broken SSL communications allowed us to perform different man-in-the-middle attacks. We were not only able to bypass the secure browsing module, but to abuse it for code execution. In our responsible disclosure process, we informed all affected security companies about our findings in their applications and to the best of our knowledge all vulnerabilities, except for one, were fixed. However, the overall study shows the fact that also big security companies are not perfect in the application development process. We argue that these apps should be reviewed and tested more thoroughly before distribution.

Acknowledgements.

The research reported in this work has been supported in part by the German Federal Ministry of Education and Research (BMBF) and by the Hessian Ministry of Science and the Arts (HMWK) within CRISP. We further would like to thank our students Michael Tröger, Andreas Wittmann, Philipp Roskosch, Daniel Magin, Joseph Varghese, and Max Kolhagen for their contributions to this project.

8. REFERENCES

[1] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., and Zimmermann, P. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 5–17.

[2] Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., et al. Drown: Breaking tls using sslv2.

[3] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 239–252.

[4] Chin, E., and Wagner, D. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications*. Springer, 2013, pp. 138–159.

[5] Enck, W., Traynor, P., McDaniel, P., and La Porta, T. Exploiting open functionality in sms-capable cellular networks. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 393–404.

[6] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 50–61.

[7] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 49–60.

[8] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

[9] Garman, C., Paterson, K. G., and der Merwe, T. V. Attacks only get better: Password recovery attacks against rc4 in tls. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 113–128.

[10] Google. Android debug bridge. https://developer.android.com/studio/command-line/adb.html.

[11] Google. Google cloud messaging. https://developers.google.com/cloud-messaging/.

[12] Google. Intent. https://developer.android.com/reference/android/content/Intent.html.

[13] Google. Intents and intent filters. https://developer.android.com/guide/components/intents-filters.html.

[14] Joxean, K. Breaking antivirus software. In *SYSCAN 360* (2014).

[15] Kantola, D., Chin, E., He, W., and Wagner, D. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 69–80.

[16] Klein, A. Attacks on the rc4 stream cipher. *Designs, Codes and Cryptography 48*, 3 (2008), 269–286.

[17] Mantin, I., and Shamir, A. *Fast Software Encryption: 8th International Workshop, FSE 2001 Yokohama, Japan, April 2–4, 2001 Revised Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, ch. A Practical Attack on Broadcast RC4, pp. 152–164.

[18] Min, B., Varadharajan, V., Tupakula, U., and Hitchens, M. Antivirus security: naked during updates. *Software: Practice and Experience 44*, 10 (2014), 1201–1222.

[19] Möller, B., Duong, T., and Kotowicz, K. This poodle bites: exploiting the ssl 3.0 fallback. *Google, Sep* (2014).

[20] Niemietz, M., and Schwenk, J. Ui redressing attacks on android devices, 2014. BlackHat Asia.

[21] Nikolay Elenkov. Unpacking android backups, june 2012. https://nelenkov.blogspot.de/2012/06/unpacking-android-backups.html.

[22] Rasthofer, S., and Arzt, S. (in-)security of backend-as-a-service solutions. In *Blackhat Europe* (Nov. 2015).

[23] Rastogi, V., Chen, Y., and Jiang, X. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 329–334.

[24] Ryan Welton. A pattern for remote code execution using arbitrary file writes and multidex applications, june 2015.

https://www.nowsecure.com/blog/2015/06/15/a-pattern-for-remote-code-execution-using-arbitrary\-file-writes-and-multidex-applications/.

[25] Ryan Welton. Remote code execution as system user on samsung phones, june 2015. https://www.nowsecure.com/blog/2015/06/16/remote-code-execution-as-system-user-on-samsung-phones/.

[26] Sedol, S., and Johari, R. Survey of cross-site scripting attack in android apps. *International Journal of Information & Computation Technology 4*, 11 (2014), 1079–1084.

[27] Siegfried Rasthofer, Irfan Asrar, S. H., and Bodden, E. How current android malware seeks to evade automated code analysis. In *9th International Conference on Information Security Theory and Practice (WISTP'2015)* (2015).

[28] Simon, L., and Anderson, R. Security Analysis of Android Factory Resets. In *Proceedings of 4th Workshop on Mobile Security Technologies (MoST)* (2015).

[29] Simon, L., and Anderson, R. Security analysis of consumer-grade anti-theft solutions provided by android mobile anti-virus apps. In *in 4th Mobile Security Technologies Workshop* (2015).

[30] Snell, B. Mobile threat report, what's on the horizon for 2016, june 2015. http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf.

[31] Tavis Ormandy. Project zero issue tracker, june 2015. https://bugs.chromium.org/p/project-zero/issues/list?can=1&q=taviso.

[32] Zane Lackey, Luis Miras. Attacking sms, 2009. http://luis.ringzero.net/docs/Attacking_SMS_-_BlackHat_2009.pdf.