(In-) Security of Security Applications

Security Advisories

Stephan Huber, Siegfried Rasthofer, Steven Arzt, Michael Tröger, Andreas Wittmann,
Philipp Roskosch, Daniel Magin, Joseph Vargehese, Max Kolhagen

This document aggregates all the security advisories, which were sent to the different application provider informing them about the vulnerability, how an attacker can abuse them and suggestion how to fix the flaws. The advisories have been published as additional information to the press release to deliver more technical details for the interested reader.

The research reported in this document 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 (www.crisp-da.de/). The work has been done by members of the Fraunhofer SIT research department Secure Software Engineering (SSE), the Fraunhofer SIT research department Testlab Mobile Security (TMS), and selected high-potential students from different universities. This cross-department research cooperation with students is organized in a team called TeamSIK (https://www.team-sik.org/).

Table of Contents

1. AndroHelm Antivirus

1.1 Summary

Vendor: AndroHelm Antivirus
Product: Virenschutz für Android App
(https://play.google.com/store/apps/details?id=com.androhelm.antivirus.free2)
Affected Version: 1.6, Platform Build Version Name 5.0.1-1624448
Severity: medium
Remote exploitable: yes

The application "AndroHelm Antivirus" by AndroHelm Antivirus contains different implementation flaws, which can be abused locally or remotely to crash the application. Furthermore, the payment verification can be abused to enable the PRO features of the application without paying the requested fee. In the context of the anti-theft feature a logic flaw allows an attacker to remotely activate that features without any authentication.
We evaluated and can confirm the applicability of the following attack vectors as successful:

a) Implementation flaws handling SMS
b) Sensitive data storage in `SharedPreferences` (missing Server verification)

Attacks based on Faulty SMS Handling
All attacks considering topic a) can be executed/ triggered remotely by sending specific SMS messages. The attacks can also be done locally by emulating the sending of an SMS or the corresponding payload. The application can be crashed and/or also the SMS blacklisting filter can be bypassed by an SMS which does not contain a regular phone number, but instead a textual string. The presented SMS number (PDU part) is not guaranteed to be a plain number, there can be also alpha numeric entries or nothing. The internal `BroadcastReceiver` and the database structure handles SMS sender numbers only as integer values. This type inconsistence can be abused for crashing the app or bypassing the SMS blacklisting filter.
See :

```
run app.broadcast.send --action android.provider.Telephony.SMS_RECEIVED --extra string test
value
```

logcat then shows a NullPointerException:

```
Java.lang.RuntimeException: Unable to start receiver
com.androhelm.antivirus.receivers.SMSMonitor: java.lang.NullPointerException
E/AndroidRuntime(16060): at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2383)
E/AndroidRuntime(16060): at android.app.ActivityThread.access$1500(ActivityThread.java:141)
E/AndroidRuntime(16060): at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1310)
E/AndroidRuntime(16060): at android.os.Handler.dispatchMessage(Handler.java:99)
E/AndroidRuntime(16060): at android.os.Looper.loop(Looper.java:137)
E/AndroidRuntime(16060): at android.app.ActivityThread.main(ActivityThread.java:5041)
E/AndroidRuntime(16060): at java.lang.reflect.Method.invokeNative(Native Method)
E/AndroidRuntime(16060): at java.lang.reflect.Method.invoke(Method.java:511)
E/AndroidRuntime(16060): at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:793)
E/AndroidRuntime(16060): at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:560)
E/AndroidRuntime(16060): at dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime(16060): Caused by: java.lang.NullPointerException
E/AndroidRuntime(16060): at
com.androhelm.antivirus.receivers.SMSMonitor.onReceive(SMSMonitor.java:31)
E/AndroidRuntime(16060): at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2376)
E/AndroidRuntime(16060): ... 10 more
```

Another attack considering topic a) are the anti-theft features. The code part (`SMSMonitor` class) contains a logic flaw: if the application does not define a friend's phone number, an attacker can use

prepared SMS messages to activate the features. In general, the message for receiving anti-theft SMS looks like this:
`SMS_PASSWORD wipe` or
`SMS_PASSWORD lock PIN_PASSWORD`
In the source-code the message of the SMS will be parsed with the `split`-function given a space symbol (`split(" ")`). The password will be checked against an empty string (since anti-theft is disabled); so we need to get an empty string as a password. This can be done by starting the SMS with a space like this:
`[SPACE]wipe[SPACE]someString`
If an attacker sends an SMS with an empty number field and the content
`[space]command[space]someString` the command will be executed.
For wiping and locking, the administration option must be set, otherwise the app crashes.

Attacks based on Storing Sensitive Data within Shared Preferences
This paragraph focusses on flaws introduced by sensitive data storage in `SharedPreferences`. The first minor flaw is the storage of anti-theft passwords, which are stored in plaintext in a `SharedPreferences` file.

```
      <map>
      <string name="prefFriend1">sd</string>
      <null name="prefFriend2" />
      <string name="antiThieftPassword">password</string>
      <null name="email" />
      <string name="prefFriendPassword">friendpassword</string>
      <string name="name">test</string>
      </map>
```

The second, but more significant attack introduced by storing sensitive data within `SharedPreferences` is targeting the PRO feature. It is feasible to activate the PRO features within the application without any valid payment. The PRO feature verification is only performed on client side. An attacker just has to set the `value=”true”` for the `“isPro”` field in com.androhelm.antivirus.free.preferences.xml file to activate all premium features. The modification of the `SharedPreferences` file can be done by a user without root privilege. The attacker can extract the file with the adb backup feature and modify the `com.androhelm.antivirus.free.preferences.xml` file by inserting the following entry:

```
      <boolean name="isPro" value="true" />
```

After that, the modified file can be restored to the device and the app is running as a full premium version.

1.2 Impact

The presented attacks are targeting the business model as well as significant security features of the app.

1.3 Workaround

For the SMS threats a user can protect himself if he defines friend’s entries and a password for the anti-theft features.
The premium activation is not threatening the user, this can be abused for causing financial loss for the vendor.
All presented flaws should be fixed on the code layer by the developer.

1.4 Fix

Verify the intent value against null in the `onReceive(Context, Intent)` method in Broadcast receivers. The SMS logic must also consider SMS messages, which contain empty or String values as senders, also the database data type.
The anti-theft password should be stored in encrypted form and the premium feature unlocking should be verified on server side.

2. Avira

2.1 Summary

Vendor: Avira
Product: Avira Antivirus Security for Android
(https://play.google.com/store/apps/details?id=com.avira.android)
Affected Version: 4.2 , Platform Build version 5.0.1-1624448
Severity: medium
Remote exploitable: yes

The Avira Android application contains implementation flaws, which can be abused by a man-in-the-middle attack or local attack. The update traffic at install time or requested by user is handled by an unprotected and unauthenticated http connection.
A man-in-the-middle attacker, controlling a WI-FI access point can modify the request/responses of the communication or redirect the traffic to another server providing manipulated files. The update process contains signature information and the application also verifies the signature of the downloaded files, but this can be partially circumvented. The following section describes how to circumvent this process.
We tested/confirmed the following attacks:

a) An attacker can exchange (downgrade) the Virus definition file

b) Overwrite a binary file (.so) with a "corrupted" file and deactivate the scan engine without user knowledge (denial of service).

c) Local denial of service attacks

Exchange/Downgrade Virus Definition File
Downgrading/replacing by an empty virus definition file can be done by redirecting the connection and providing a modified `axvdf-common-int.info`. By changing the module (<FILE> - Tag entry) containing the file information (md5sum, zip md5sum, file size, etc.) referencing for instance an empty .vdf file, the updater will download the provided empty files. The updater currently only consider the md5 values of the files, which does not provide a reliable integrity protection. The modified files and also additional new entries will not be verified by the integrity check. The additional files will be downloaded from the redirected server into the temp folder afterwards written into the `/data/data/com.avira.android/bin/antivirus` folder. The scan engine and also the user interface will not provide any error message to the user.

Deactivate Avira Scan Engine
An attacker can deactivate the Avira scan engine (`libaescn.so` file) by overwriting it with a non-binary file. The engine will not run any longer and will deliver an error to the log interface:

```
E/Avira Mobile Security( 2257): Antivirus - Failed to initialize scan
engine - Error = 121
E/Avira Mobile Security( 2257): Antivirus - Rolling to backup
```

The user interface on the application layer does not interpret or handle error messages from the scan engine layer. As a result the application shows still a scanning task to the user without any critical findings.
The overwriting of the `libaescn.so` by an attacker can be done as follows:
Because of a missing integrity check of the `aengine-arm_linux_androideabi-int.info` file <FILE> an attacker can add an additional <FILE> section:

```
   <FILE>
        <NAME value="axvdf/common/int/libaescn.so"/>
        <FILEMD5 value="93dbbc5000427a4f513a05480b5ff1a6"/>
        <PEFILEMD5 value="93dbbc5000427a4f513a05480b5ff1a6"/>
        <FILESIZE value="1228"/>
        <ZIPMD5 value="e707a871b70eba16482be4374e94a329"/>
        <ZIPSIZE value="477"/>
        <OS value="ALL"/>
        <VERSION value="8.1.8.4"/>
  </FILE>
```

In this scenario the injected libaescn.so file will be loaded a second time and overwrite the original `libaescn.so` binary.
The Update engine checks the signature of the downloaded .so file, so the injection of a remote Binary (remote code) injection is currently not working. But if we use for instance another signed Avira file with textual content like the `aevdf.dat` file and rename it to `libaescn.so` it will bypass the signature verification and overwrite the `libaescn.so`, previously loaded with textual content. After that the native code layer cannot load the file as binary anymore and returns the `Error = 121` code.
It is also possible to locally exchange the libmavapi.so file by another native code binary, which will be executed instead by the scan engine. But currently we did not find a solution to inject this file remotely or by another app without breaking the sandboxing (e.g. root application).

Local Denial-of-Service Attacks
A local denial of service attack (crash the Avira app) can be caused by sending an empty SMS broadcast. The command can look like:

```
am broadcast -a android.provider.Telephony.SMS_RECEIVED
```

This happens because of missing null value checks in the broadcast receiver in the class `BLOnSmsBroadcastReceiver.class` .

```
public void onReceive(Context arg7, Intent arg8) {
        Bundle v0 = arg8.getExtras();  <== //Problem
        if(v0 != null) {
            Object v0_1 = v0.get("pdus");
```

A malicious application can use this to crash the app and prevent the execution of the application. Newer Android Versions (since Lollipop) mitigate such null value broadcast, but in pre Lollipop versions it is possible to crash the app.

2.2 Impact

Depending on the target of the attacker he can abuse this for remotely deactivating the Avira scan and protection mechanism. In a second step he can try to infect the victim with malicious code.

2.3 Workaround

Avoid public or unknown Wi-Fi hotspots (access points) or use them only through a trusted VPN connection.

2.4 Fix

Proposals of a vendor fix can be done using different approaches: (1) transfer the file(s) using a protected SSL (HTTPS) connection and/or (2) establish a correct signature process for the configuration files and the downloaded files to prevent modification of the files.

3. Cheetahmobile

3.1 Summary

Vendor: cheetahmobile
Product: CM Security
(https://play.google.com/store/apps/details?id=com.cleanmaster.security)
Tested Version: 2.7.3 and 2.8.5
Severity: medium-high
Remote exploitable: yes

The CM Security Android application contains different implementation flaws, which can be abused by an attacker. The application is vulnerable to man-in-the-middle (mitm) attacks and different types of tap jacking attacks. Additionally there is an exported Service (`ks.cm.antivirus.defend.DefendService`) that might be abused by local apps because of exporting the service without any permissions (not tested in detail).

3.2 Impact

Mitm attack, code injection and signature file downgrade:
The virus definition update process of the application is done via an unprotected http connection. A man-in-the-middle attacker controlling a WI-FI access point can modify the request/responses of the communication or redirect the traffic to another server (faking an update server) providing manipulated files. Attack process:

1) In the first step the application loads the `version_data.ini` file which contains the update version information and the path of the `info_data.ini` file (containing further file information). The info_data.ini file is stored in a `.zip` file named with the md5 value of the .zip file (e.g. `c843c25d0ef76eb3bd5122db547f1656.ini.zip`).

All files and information stored in the config files mentioned in 1 can be spoofed by an attacker.

2) The `info_data.ini` file contains further paths and files which will be downloaded during the update process. These files are for instance data base files containing firewall rules, virus signatures, exploit pattern or .so (native executable) files. The info_data.ini file has no integrity protection, thus an attacker also can change and manipulate the content of this file.

The missing integrity protection can be abused by attackers replacing the `.db` files through empty or manipulated files. The security app won't detect malicious apps any more.
The biggest danger in the mitm attack scenario is the loading of native binaries. An attacker can replace (inject) a manipulated binary file. In the version 2.7.3 and previous versions we could observe the following download:
http://dl.sj.ijinshan.com/duba/updateintl/data/2015.11.26.2058/armeabi/libavlm.so.zip
Adapting the md5 and size values in the `info_date.ini` file triggers the download of a manipulated binary from a fake update host.
An attacker can replace the binary (libavlm.so), which is downloaded with the update process, by another binary file. This file will be executed by the application.
In the last tested version (2.8.5) we were not able to download the binary.
The traffic redirection or traffic payload manipulation is possible because the connection has no integrity protection and the client (application) is not authenticating the server.

Tapjacking Attack:
The application exports a lot of activates and app components which can be started externally by third party apps. A malicious app can start a component of the app, put an overlay over the component and trick the user to deactivate security features of the application.

For further details about tapjacking and protection mechanism look at:
https://media.blackhat.com/ad-12/Niemietz/bh-ad-12-androidmarcus_niemietz-WP.pdf

3.3 Workaround

Against mitm Attack:
Avoid public or unknown Wi-Fi hotspots (access points) or use them only through a trusted VPN connection.

3.4 Fix

Against mitm Attack:
Proposals of a vendor fix can be done by different approaches: (1) transfer file(s) with a protected SSL (HTTPS) connection and/or (2) establish a correct signature process of the configuration files and the downloaded files to prevent modification of the files.

Against tapjacking:
Reduce exported application and implement tap jacking protection. The Android API provides different methods to mitigate tapjacking (see:
http://developer.android.com/intl/es/reference/android/view/View.html#setFilterTouchesWhenObscured%28boolean%29)

4. ESET

4.1 Summary

Vendor: ESET
Product: Mobile Security & Antivirus
(https://play.google.com/store/apps/details?id=com.eset.ems2.gp)
Affected Version: 3.2.4.0, Platform Build version 4.2.2-1425461
Severity: medium to high
Remote exploitable: yes

The ESET Mobile Security & Antivirus application contains implementation flaws which can be abused by an attacker. The application contains an incorrect SSL certificate validation, which can be abused by an attacker for eavesdropping or manipulating the communication. Furthermore, an attacker can steal the ESET license information or the user and password for the license.
The *Mobile Security* scanner also implements an encryption scheme, which is violating Kerkhoffs's principle (hide the key, not the algorithm). The implemented crypto scheme is vulnerable to chosen cipher text attacks and could be easily broken. Encrypted information can be decrypted to plaintext. We wrote a short proof of concept script attached at the end of this document.
The crypto scheme is implemented within the native code layer. It should be verified if the scheme is utilized in other ESET products. If this is the case, please fix it also there.

4.2 Impact

Incorrect SSL Certificate Validation:

NOTE: Due to the obfuscation of the application, we do not mention unobfuscated class or method names here. Therefore we refer to obfuscated class or method names only.
The class `jl` implements the `X509TrustManager` interface, which defines the `checkServerTrusted` method, handling the server certificate validation. In the current implementation this method is empty, which means it does not validate the correctness of the server certificate nor the whole certificate chain. A Man-in-the-Middle attacker, e.g. at an open Wi-Fi Hotspot, can eavesdrop the SSL traffic because the application will trust all certificate. It is not necessary to install a compromised root certificate on the device. Each communication using the faulty `Trustmanager` can be compromised.
The following excerpt was derived from the registering process traffic (captured using a Man-in-the-Middle attack):

```
<?xml version="1.0" encoding="UTF-8"?>
<REGISTERING>
<SECTION ID="1000103">
<REGISTERINGREQUEST>
<NODE NAME="UidType" VALUE="8" TYPE="DWORD"/>
<NODE NAME="CountryCode" VALUE="US" TYPE="STRING"/>
<NODE NAME="EmailAddress" VALUE="" TYPE="STRING"/>
<NODE NAME="Note" VALUE="[LGE] [occam]" TYPE="STRING"/>
<NODE NAME="Platform" VALUE="Android 4.4.4" TYPE="STRING"/>
<NODE NAME="Product" VALUE="715" TYPE="STRING"/>
<NODE NAME="Version" VALUE="3.0.1318.0" TYPE="STRING"/>
<NODE NAME="LocaleCode" VALUE="de" TYPE="STRING"/>
<NODE NAME="EvCode" VALUE="eeXrRK5Ipltsamy8SshZ1di0Jvo=" TYPE="STRING"/>
<NODE NAME="CustomCode" VALUE="0" TYPE="DWORD"/>
<NODE NAME="ICode" VALUE="MDAwMDAwwCxwXphYj3JMoEasWcr+zmVQHjY="
TYPE="STRING"/>
<NODE NAME="LicenseUsername" VALUE="Fdax6a7wj/I+ZEet" TYPE="STRING"/>
<NODE NAME="LicensePassword" VALUE="Fdax6a7wj/I=" TYPE="STRING"/>
```

```
<NODE NAME="PreviousLicenseUsername"
VALUE="JNaV6Yvw1vJrZAStgsgqddgxE7x6+OkMc9013Q==" TYPE="STRING"/>
<NODE NAME="PreviousLicensePassword" VALUE="VNa26a/wzvIjZAetwsgtdY4xGLw="
TYPE="STRING"/>
</REGISTERINGREQUEST>
</SECTION>
</REGISTERING>
```

A Man-in-the-Middle attacker can steal the username and password for the AV license (see bold information in listing above). The given encryption/obfuscation, behind the BASE64 encoding, of the credential information does not provide enhanced protection in this attack scenario, because the attacker does not need to decrypt it to authenticate. He retransmits the sniffed (partial) XML information in his own context and gets authenticated.
An additional attack scenario would be to decrypt the credentials and abuse it for something else, like additional ESET services. This can be done because the implemented encryption scheme can be easily broken by chosen plain text attacks. Further details can be found in the next section.

Crypto Scheme, Chosen Cipher Text Attack and Decryption Algorithm:
In our analysis we did not completely reverse engineer the applied encryption scheme for the license information, but we used chosen cipher text attacks in order to break it.
The encryption is done by XOR the input value with a key generated by a random stream. In order to retrieve the key we utilized, for instance a username consisting of 25 times "a". After that the application encrypted the username and the application with BASE64. We decoded the BASE64 String and XORed the byte array (50 byte) with 25 times of "a" (each 2nd byte) and we got a key with the length of 25 bytes.
We attached a proof of concept script, which will decrypt the `LicenseUsername` and `LicensePassword` up to the length of 25 characters.
In the same way it would also be possible to generate longer keys but it depends of the length of username and password. Most people choose shorter ones.

4.3 Workaround

App users should avoid public unknown (unreliable) Wi-Fi Hotspots because Man-in-the-Middle attacks are possible (transparent proxy, ARP spoofing etc.). Data transfer over GPRS/UMTS is more secure because it requires higher effort to do a Man-in-the-Middle attack, but cannot be excluded. If public Wi-Fi connections are utilized, a VPN tunnel to a trusted counterpart should be established.

4.4 Fix

The SSL verification implementation has to be fixed. In general, it is not a good idea to overwrite the `TrustManager` or perform certificate verifications using own implementations. The operation system functions will handle SSL verification properly by default. For best practices, see the Android Developers references (http://developer.android.com/intl/es/training/articles/security-ssl.html).
For higher security demands, please refer to certificate pinning. But this approach must be evaluated if it is compliant to the backend infrastructure.
The encryption scheme should be avoided or removed. Self-implemented crypto solutions are usually insecure. For the context of authentication, the encryption of the license information does not bring any benefit. The Man-in-the-Middle attacker must only forward the stolen data, independent if it is encrypted or not. If the SSL validation is implemented correctly, the credentials are encrypted implicitly.
If you aim to protect the credentials additionally, because someone can abuse it for other services if he gets them in plaintext, we recommend to use a hash function instead of an encryption function. To protect the hash function against rainbow tables augment them with a salt value.

APPENDIX

Python Script for decrypting username or passwords. The POC contains a 25 byte long key for showing how simple the encryption scheme can be broken.

```
#!/usr/bin/python2.7
"""
Created on Mon Nov 30 16:39:39 2015
@author:
"""

key = [97, 212, 221, 251, 91, 53, 183, 25, 236, 43, 66, 217, 75, 7, 198, 8,
17, 81, 212, 184, 191, 169, 203, 45, 123]

def decrypt(inString, key):
    inString = bytearray( inString.decode("base64") )   # decode as base64
    inString = inString[::2]                            # take every second
char
    result = list()
    for i, c in enumerate( inString ):
        cipher = c ^ key[i]
        result.append( chr(cipher) )
    return result

print("Type 'q' to quit")

while True:
    inputString = raw_input("Enter base64 string to decrypt: ")
    if inputString == "q":
        exit()

    try:
        res = decrypt( inputString, key )
        print("Input: " + inputString)
        print("Encrypted: " + ''.join(res) + "\n")
    except:
        print("* Cannot decode base64 string!")
```

5. Kaspersky

5.1 Summary

Vendor: Kaspersky
Product: Kaspersky Internet Security for Android
(https://play.google.com/store/apps/details?id=com.kms.free)
Affected Version: 11.9.4.1294
Severity: medium - high
Remote exploitable: yes

The Android app named “Kaspersky Internet Security” downloads advertisement information using unprotected HTTP connection (`http://ipm.kaspersky.com/600eb07a-2926-4407-b014-d3e8c77b0086.zip` and `http://ipm.kaspersky.com/eeea9321-5eac-4709-9046-8475ee951c82.zip`). The downloaded zip file contains some HTML and CSS files utilized for presenting advertisement information to the user.
Due to the unprotected HTTP connection, a n attacker located can manipulate the transmitted zip files using a man-in-the-middle attack. The Kaspersky Internet Security app extracts the content of the zip file(s) to `/app_ipm/` folder within the app’s directory on the Android smartphone. If one of the (attacker manipulated) zip file(s), which has been automatically downloaded by the app, contains an additional or manipulated pdm.jar file, the file will be extracted by the app. To force the app to execute the manipulated pdm.jar file, the original pdm.jar file in the `/app_bases/` folder has to be overwritten. In order to do this, the unzip (extraction) algorithm can be abused to perform path or directory traversing in order to store the extracted file in the attacker selected directory (`/app_bases/` folder) in contrast to the designated directory (`/app_ipm/` folder).
This is the case, if the zip file contains a file with the full name
`“../../../../../../../../../../../../../../../../../../../../../../data/data/com.kms.free/app_bases/pdm.jar”`
The original `pdm.jar` in the `/app_bases/` folder will be overwritten and the manipulated `pdm.jar` (contains `.dex` file) will be executed from the Kaspersky Internet Security app.
The man-in-the-middle attack was done with the help of a modified Wi-Fi access point running a transparent proxy. With additional equipment it is also thinkable to do such an attack on a GSM or UMTS data connection.

5.2 Impact

Depending on the target of the attacker he can abuse this for injecting additional code into the application. The injected code will run in the (sandboxed) context of the Kaspersky Internet Security app. But the Kaspersky Internet Security app has a magnitude of permissions and the executed code can thus use a wide range of API calls. The loaded code may also execute a root exploit to escalate privileges and to establish some root (remote) shell.
An additional (theoretical) scenario, an attacker may hijack the advertisement providing server and he may abuse it to spread (remote) code to all Kaspersky apps and establish a botnet.

5.3 Workaround

Avoid public or unknown Wi-Fi Hotspots (access points) or use them only through a trusted VPN connection.

5.4 Fix

Proposals of a vendor fix can be done using different approaches: (1) transfer the zip file(s) using a protected SSL (HTTPS) connection and/or (2) include the unprotected traffic to the integrity protection within the app (similar to the signature and root detection updates the app retrieves from Kaspersky’s servers).

6. Malwarebytes

6.1 Summary

Vendor: Malwarebytes
Product: Malwarebytes Anti-Malware
(https://play.google.com/store/apps/details?id=org.malwarebytes.antimalware)
Affected Version: versionName="2.00.3.9000", platformBuildVersionName="5.1.1-1819727"
Severity: medium
Remote exploitable: yes

The Malwarebytes application contains implementation flaws which can be abused by a man in the middle attack or local attacker. The update traffic at install time or requested by user is handled by an unprotected and unauthenticated http connection.
A man-in-the-middle attacker, controlling a WI-FI access point can modify the request/responses of the communication or redirect the traffic to another server providing manipulated files. The files are encrypted with symmetric keys, stored locally in the application.
Another weakness is a faulty SSL implementation (certificate check).
Furthermore the app can be crashed by a local broadcast message.

Unprotected Update

The update mechanism via http has no authentication and integrity protection. A man-in-the-middle attacker can change or manipulate the traffic. The signature file is protected by a static key (“**TI028Z%th5Y'uX4>dQz|26ULy+.z5$;w)YG2G8ac!9\"{T!W*lp92k@CltD29+(W**”) hardcoded in the application and compressed with gzip algorithm. The zlib compressed phishing signatures File is encrypted with the md5 sum of the static key (“**xf4kfh9dlk483jmfbn5ujf8dj46m7i8fj4mf96adfh543**”). An attacker who extracts the key can manipulate the update files.
Furthermore the encryption algorithm RC4 is an algorithm which should not be used any more.

Faulty SSL Implementation

In the class `ara` (bytecode name because of class name obfuscation) the `checkServerTrusted` method is overwritten and does not verify the SSL certificate chain in a correct manner.

```
…
public void checkServerTrusted(X509Certificate[] chain, String s){
int v1 = chain.length;
for(int v0 = 0; v0 < v1; ++v0) {
            chain[v0].checkValidity();
        }
}
…
```

The method only verifies the validity Date of the certificate chain (Date valid before and after). (see: http://developer.android.com/intl/es/reference/java/security/cert/X509Certificate.html#checkValidity%28%29). This means a mitm attacker can use each type of self-signed certificate with a valid date to eavesdrop or manipulate the SSL connection.

Local Denial-of-Service Attacks

A local denial of service attack (crash the Malwarebytes app) can be triggered by sending an empty install package broadcast.
The command can look like:

```
am broadcast -n
org.malwarebytes.antimalware/org.malwarebytes.antimalware.security.scanner.
receiver.ScAppInstallReceiver
```

Log output of Runtime Exception:

```
E/AndroidRuntime(18990): Process: org.malwarebytes.antimalware, PID: 18990
E/AndroidRuntime(18990): java.lang.RuntimeException: Unable to start receiver
org.malwarebytes.antimalware.security.scanner.receiver.ScAppInstallReceiver:
java.lang.NullPointerException: Attempt to invoke virtual method 'boolean
java.lang.String.contains(java.lang.CharSequence)' on a null object reference
E/AndroidRuntime(18990):      at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2616)
…
E/AndroidRuntime(18990): Caused by: java.lang.NullPointerException: Attempt to invoke virtual
method 'boolean java.lang.String.contains(java.lang.CharSequence)' on a null object reference
E/AndroidRuntime(18990):      at
org.malwarebytes.antimalware.security.scanner.receiver.ScAppInstallReceiver.onReceive(org.malw
arebytes.antimalware.security.scanner.receiver.ScAppInstallReceiver:39)
E/AndroidRuntime(18990):      at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2609)
E/AndroidRuntime(18990):      ... 10 more
```

This happens because of missing `null` value checks in the broadcast receiver in the class `ScAppInstallReceiver.class`.

```
…
public void onReceive(Context arg5, Intent intent) {
     super.onReceive(arg5, intent);
     if(PreferenceUtils.a(arg5, 2131100041)) {
        String v0 = intent.getDataString(); //<= Problem, intent can be null
            if(v0.contains("package:"))
…
```

A malicious application can use this to crash the app. Newer Android versions (since Lollipop) mitigate such null value broadcast (try to restart the app), but in pre Lollipop versions it is possible to crash the app completely.

6.2 Impact

Depending on the target of the attacker he can abuse the vulnerabilities for different purpose. The crypto flaw can be abused for manipulating and downgrading the detection engine.
The faulty SSL implementation allows eavesdropping or manipulating the transferred data.
The DOS attack reduces the availability of the application and the real-time detection.

6.3 Workaround (optional recommendation for end user)

Avoid public or unknown Wi-Fi hotspots (access points) or use them only through a trusted VPN connection.

6.4 Fix

Proposals of a vendor fix can be done using different approaches: (1) transfer the file(s) using a protected SSL (HTTPS) connection and/or (2) establish a correct signature process of the update files/ downloaded files to prevent modification of the files. Avoid unsecure local crypto key storage in the application. The unprotected symmetric approach must be removed from the application.
Using SSL communication, the `TrustManager` implementation should not overwrite the `checkServerTrusted` method. The Android operating system is implicitly doing correct certificate validation. If it is required to overwrite the certificate check, this should be done in a correct manner.

6.5 Remark

A proof of concept code (python script) decrypting and unpacking the signature files can be delivered on demand.

7. McAfee (Intel Security)

7.1 Summary

Vendor: McAfee (Intel Security)
Product: Security & Power Booster -free
 (https://play.google.com/store/apps/details?id=com.wsandroid.suite)
Affected Version: 4.5.0.601, Platform Build version 5.0.1-1624448
Severity: medium - high
Remote exploitable: yes

The McAfee Android Security application contains implementation flaws, which can be abused by an attacker. The application, in particular the *Secure Browsing* module contains an XSS-scripting vulnerability. Furthermore, an implementation flaw exists which can be used by a local malicious application to stop/kill the security app.

7.2 Impact

XXS-Scripting

The *Secure Browsing* feature should protect the user from malicious web sites. If the user enters such a malicious web page the browsing module tries to block the page by showing the McAfee Site advisor LIVE protection alert. This alert page contains a XSS-vulnerability which allows one to embed and execute JavaScript code.
See code example

```
<script type="text/javascript">

var lksu = "http://team-sik.org/";alert("XSS");var x="";

if (lksu != "") {
   document.getElementById('gobackbtn').href = lksu;
}
else {
   if (history.length <= 1) {
      divelem = document.getElementById("gobackbtn")
   if (divelem) {
      divelem.style.display = "none"
   }
}
```

The blocking page shows the URL of the calling site and embeds it into the blocking page. If the URL contains JavaScript it will be executed (see bold line). An attacker or malicious site can inject JavaScript to bypass the blocking page or execute other arbitrary JavaScript code.
We evaluated this on the Android Application, but we assume if McAfee uses this approach also within the desktop version; then this XSS-scripting vulnerability would be also affect desktop systems.

The related code within the Android app for this vulnerability is located mostly in `com.mcafee.android.siteadvisor.service.d.` `Method a(final String, int, final ComponentName)` sets the last known safe URL:

```
[...]
if (this.h) {
    if (i.a("SiteAdvisorManager", 2)) {
        b.d("Inside onPageLoad call to set Last KnowSafeURL" + a2);
    }
    if (obj != null) {
        this.g = a2;                              <=== saves last known URL
        if (i.a("SiteAdvisorManager", 2)) {
            b.d("Setting Last KnowSafeURL onPageLoad : " + this.g);
        }
    }
} else if (!a.a.e) {
    this.g = a2;                                  <=== saves last known URL
    if (i.a("SiteAdvisorManager", 2)) {
        b.d("Setting Last KnowSafeURL directly : " + this.g);
    }
} else if (!a2.equals(this.g)) {
    if (i.a("SiteAdvisorManager", 2)) {
        b.d("Calling ResolveLastKnownSafeURL to set Last KnowSafeURL" +
a2);
}
[...]
```

As a proof of concept code we provide a sample link from the Secure Browsing feature showing the JavaScript injection (JavaScript must be enabled in Browser).
http://www.sa-live.com/mprot2.html?v=1&ui=0&spid=mcafee&px=00080000000000000000000000000000&c=0x821&url=2RYXRrK9YAOqKU0G8LVTS4HB%2BI83e7Oi&vascheme=com.wsandroid.suite&sabp=true&langcode=en-US&lksu=2RYXRrK9YAaNuGgMTPebqRFBVEM0gqIkolCJFeYHVin2OYs33qsYmg%3D%3D&enc=t#

Denial-of-service Attack:
The application contains a Baidu plugin which does not properly handle received intents containing Null values. Android OS before Lollipop (Version < 5) have no Null intent value filtering, so if an application sends a Broadcast Intent with an empty value, the BroadcastReceiver of the McAfee application cannot handle this and crashes.
Example Intent generated by drozer[1] tool:

```
run app.broadcast.send --action
com.baidu.android.pushservice.action.RECEIVE --component
com.wsandroid.suite com.mcafee.messaging.baidu.BaiduMessageReceiver
```

App crashes because of a `NullPointerException` completely. Intent has to be sent 3 times to ensure that the app will not restart again.
Responsible Code (Code where crash happens):
Class `com.baidu.frontia.api.FrontiaPushMessageReceiver`

```
public final void onReceive(Context context, Intent intent) {
   int i = 0;
   [...]
   } else if (intent.getAction().equals("com.baidu.android.pushservice.action.RECEIVE") ||
intent.getAction().equals(b)) {
       String stringExtra = intent.getStringExtra("method");  <==== PROBLEM IF NO EXTRA IS SET
       int intExtra = intent.getIntExtra("error_msg", 0);
       Object obj = "";
       if (intent.getByteArrayExtra("content") != null) {
           obj = new String(intent.getByteArrayExtra("content"));
       }
```

A malicious application can abuse this to crash/deactivate the McAfee application and bypass the protection on the smartphone.

[1] https://www.mwrinfosecurity.com/products/drozer/

7.3 Workaround

Deactivation of the *Secure Browsing* option.

7.4 Fix

Introduce some sanitizer which will prevent JavaScript in URL or filter URL appending and show only the base URL as string.